

I

The Design of GrIDS:
A Graph-Based Intrusion Detection System

Steven Cheung, Rick Crawford, Mark Dilger,
Jeremy Frank, Jim Hoagland, Karl Levitt,
Stuart Staniford-Chen, Raymond Yip, Dan Zerkle

*Department of Computer Science,
University of California at Davis,
Davis, CA 95616*

May 14, 1997

Abstract

Abstract here. Abstract here.

Contents

1 Overview of GrIDS	4
1.0.1 Types of Network Attack	4
1.1 GrIDS-Graph-Based Intrusion Detection System	4
1.1.1 A Simple Example	4
1.1.2 Graph Building	5
1.1.3 Aggregation	5
1.1.4 Policy	6
1.1.5 Limitations	6
2 The Graph Building Engine	8
2.1 Introduction	8
2.1.1 Goals	8
2.1.2 Overview	8
2.2 Graphs	8
2.3 Graph Engine Rules	8
2.3.1 Rulesets	9
2.3.2 An Overview of How the Rules Work	9
2.3.3 Graph Rules - A Simple Example	9
2.3.4 Auto-Computed Attributes	11
2.3.5 Rule Primitives	12
2.3.6 Importing Functions into the Graph Rules	12
2.3.7 Rule Grammar	13
2.4 Aggregation of Graphs	14
2.4.1 Introduction to Aggregation	14
2.4.2 Aggregating Attributes	15
2.4.3 When Reports Are Sent to the Parent Engine	15
2.4.4 Graph Aggregation Details	15
2.4.5 Graph Aggregation Example	16
2.5 Buffers	17
2.6 Alerts	18
2.7 Query language	19
2.8 Debugging	19
2.8.1 Introduction To Log Files	19
2.8.2 Reconstructing Behavior From Log Files	20
2.8.3 Global Log File	20
2.8.4 Ruleset Log File	21
2.9 Controlling the Graph Engine	22
2.10 The Graph Engine Algorithm	22
3 Basic Communication Protocols	24
3.1 GrIDS common packet format	24
3.2 The Graph Language	25
3.2.1 Labeling Nodes and Edges	25
3.2.2 Reports Containing a Connection	25

3.2.3	Reports Containing a Host	25
3.2.4	Reports Containing a Graph	25
3.2.5	Representing Attributes on Nodes, Edges and Graphs	26
3.2.6	Quoting in the Graph Language	26
3.2.7	Graph Grammar	26
4	The Data Source Library	28
4.1	Communication with User Interface	28
4.1.1	Control Variables	28
4.1.2	Module Controller	29
4.1.3	Protocol	29
4.1.4	Special Aliases & Subcommands to Module Controller	30
4.2	Data Source To Aggregator Communication	31
4.3	Library Functions	31
4.4	Detailed Design of Module Controller	31
4.4.1	Startup and Trust Issues for Module Controllers	31
4.4.2	Special Files for Module Controller	32
4.4.3	Machine Access Control	32
4.4.4	Access Control for Special Modules	32
4.4.5	Module Controllers Should Use Standard Port	33
4.4.6	Deleting Hosts Running Critical Modules	33
4.4.7	What Software is Executing?	33
4.4.8	Launching Recurring Modules	33
4.5	Initialization of Modules	33
5	Control of Software	34
5.1	Overview	34
5.2	Software Managers	34
5.3	Access Control	34
5.3.1	User Access Control	34
5.3.2	Machine Access Control	35
5.4	Trust Issues for Software Managers	35
5.5	Trust Issues for Module Controllers	36
5.6	IMPLICATIONS for GET/SET FORMAT	37
5.7	IMPLICATIONS for Ruleset Updating	37
5.8	Deleting Hosts Running Critical Modules	38
5.9	Software Control Protocol	38
5.9.1	Set Host Variable (header <i>shv</i>)	38
5.9.2	Get Host Variable (header <i>ghv</i>)	38
5.9.3	Set Dept Variable (header <i>sdv</i>)	38
5.9.4	Get Dept Variable (header <i>gdv</i>)	39
6	The Organizational Hierarchy	40
6.1	Introduction	40
6.2	An example	40
6.3	The Organizational Hierarchy Server	41
6.4	Organizational Hierarchy Messages	42
6.4.1	Introduction	42
6.4.2	Packet Format	42
6.4.3	Transaction Types	42
6.5	The View Serial Number Mechanism	45

7	The Network Monitor	46
7.1	Assumptions and Design Objectives	46
7.2	Events	46
7.3	What to sniff for	46
7.3.1	TELNET	47
7.3.2	NFS	47
7.4	Sniffer Control Messages	48
7.4.1	Start Up and Shut Down	49
7.5	Glossary	49
7.6	Suggestions to the Implementators	49
7.6.1	Tcpdump Argument "Expression"	49
7.6.2	Tcpdump Example Output	50
7.6.3	Implementation Plan	50
8	Network Access Policies	51
8.0.4	Language Syntax	52
9	Debugging Facilities	53
9.1	Overview	53
9.2	Central logging facility	53
9.3	Ruleset debugging servers	53
9.4	Log browsing	53
9.5	Serving log browsing requests	54
9.6	Control variables	54
9.7	Access Control	54
9.8	Requirements for others	55
9.9	Forward data flow tracing	55
9.9.1	Overview	55
9.9.2	Specifying the target connection	55
9.9.3	Forward-tracing debugging algorithm	55
9.9.4	Forward tracing messages	55
9.10	Backward data flow tracing	56
9.10.1	Overview	56
9.10.2	Backward-tracing debugging algorithm	56
10	The User Interface	57
10.1	Logging In	57
10.2	Managing the hierarchy	57
10.2.1	Add Host	57
10.2.2	Add Dept	57
10.2.3	More transactions	58
10.3	Managing rulesets	58
10.4	Alerts	58
10.5	Managing Software	58
10.5.1	Department Inspector	60
10.5.2	Host Inspector	60
10.5.3	Module Inspector	60
A	Tracing Using GrIDS	61
A.1	Introduction	61
A.2	General Considerations	61
A.3	GrIDS Query Language	61
A.4	On-Host Tracing	61
A.5	Thumbprinting	62

List of Figures

1.1	The beginning of a worm graph	5
1.2	A more extensive view of the same worm.	5
1.3	A graph amongst several departments. The dashed lines are departmental boundaries.	6
1.4	The corresponding reduced graph.	6
2.1	Two machines connecting in the production department.	14
2.2	Connection from production to management.	14
2.3	The production department graph engine's view of the same situation.	14
2.4	The view according to the engine for Acme Inc.	15
2.5	Connection between ACME and Widgets.	15
2.6	View at .com domain	15
2.7	ACME view of connection to Widgets.	15
2.8	Graph Spaces At Host Level	16
2.9	Graph Spaces At Aggregated Level	16
2.10	Example graph aggregation attribute path (time 1)	17
2.11	Example graph aggregation attribute path (time 2)	17
2.12	Example graph aggregation attribute path (time 3)	17
2.13	Example graph aggregation attribute path (time 4)	18
2.14	Connection between host a in department A and host b in department B.	18
4.1	Data flow through data source modules	28
4.2	Operation of module controller	29
6.1	An example hierarchy. Department G is about to be moved from under department E to under department D. An interface and the organizational hierarchy server are also shown.	40
6.2	The hierarchy packet format. The separator, character 254, is shown in black.	42
7.1	UDP sessions.	47
10.1	Hierarchy window after clicking on CS department icon.	57
10.2	Main alert window displaying icons for each rule set. Rule sets which have generated alerts are shown in yellow and red.	58
10.3	Window resulting from clicking on Sweep in main alert window.	58
10.4	Window after clicking on history bar.	59
10.5	Inspecting the GrIDS system for a department.	59
10.6	Inspecting the GrIDS system on a particular host.	60
10.7	Inspecting a particular module.	60

List of Tables

3.1	Allowable headers in Grids packets	24
7.1	TELNET events.	47
7.2	NFS events.	48

Chapter 1

Overview of GrIDS

1.0.1 Types of Network Attack

Here we discuss examples of attacks which have interesting large scale structure which GrIDS will detect.

A *sweep* involves a single host systematically contacting many others in succession. *Doorknob rattling* is a sweep attack that checks for poorly secured or configured hosts, (e.g. checking for weak or default passwords). The Security Administrator's Tool for Analyzing Networks (SATAN) [?] is an example of a publicly available sweeping tool that scans for vulnerabilities. There are legitimate reasons for sweep activity (SNMP polling, centralized backups). However, legitimate sweeps tend to be highly circumscribed and regular in their nature; the source host, services used, hosts contacted, and time of day may all be predictable.

Coordinated attacks are multi-step exploitations using parallel sessions where the division of steps between sessions is designed to obscure the unified nature of the attack or to allow the attack to proceed more quickly. A simple example is several simultaneous sweep attacks from multiple sources. In another example, one user makes some high visibility attacks on a set of military computers as a diversion while another attacker breaks into one of the hosts and installs Trojan horse software. The combined nature of the attack can be apparent if the actual source of the attack is traced back to the same person, or if features of the attacks are similar. To make this inference, a mechanism must exist to correlate sessions across several hosts.

Seely defines a *worm* as "a program that propagates itself across a network using resources on one machine to attack other machines" [?]. The best known worm attack is the Morris worm of 1988 which infected thousands of hosts throughout the Internet, rendering them unusable. Worms are evidenced by a large amount of traffic forming a tree-like pattern and by similar activity occurring on hosts within this tree. Intrusion detection systems may detect a worm by analyzing the pattern of spread. It would be difficult to note the presence of a worm simply by looking at a single host since the larger, widespread nature of the attack would not be apparent.

1.1 GrIDS-Graph-Based Intrusion Detection System

We now present the design of GrIDS. We begin with a very simplified version, and then explain more details.

1.1.1 A Simple Example

GrIDS will construct graphs which represent hosts and activity in a network. Let us take the tracking of a worm as an example of building such an *activity graph*. In Figure 1.1, the worm begins on host *A*, then initiates connections to hosts *B* and *C* which cause them to be infected. The two connections are reported to a GrIDS module, which creates a new graph representing this activity and records when it occurred. If enough time passes without further activity from hosts *A*, *B*, or *C*, then the graph will be forgotten. However, if the worm spreads quickly to hosts *D* and *E*, as in the figure, then this new activity is added to the graph and the graph's time stamp is updated. This simple procedure assumes that activities between machines are related if they occur closely together in time. Further activity by the worm results in an even larger graph, as in Figure 1.2. In general, when a worm infects a network protected by GrIDS, the fanning, tree-like structure of the worm's propagation will cause GrIDS to build a fanning, tree-like graph. GrIDS evaluates this graph pattern as a suspected worm. This evaluation can be performed, for example, by counting the number of nodes and branches in the graph. Counts over a threshold provoke GrIDS to report a suspected worm.

Similarly, network sweeps and other patterns of abuse produce graphs of a certain shape, and GrIDS may be configured to detect and report them.

In verifying our design concept, we built a basic implementation of this algorithm (which we christened *Early Bird*). While it would be premature to quantitatively evaluate this version, we did run the code for several weeks on our LAN with TCP-wrapper data as input. We had no difficulty in tuning the software to detect a worm or sweep attack within seconds but produce only one or two false alarms per day.

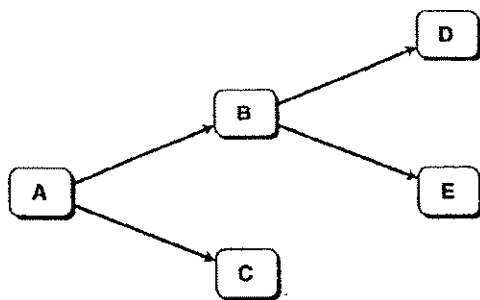


Figure 1.1: The beginning of a worm graph

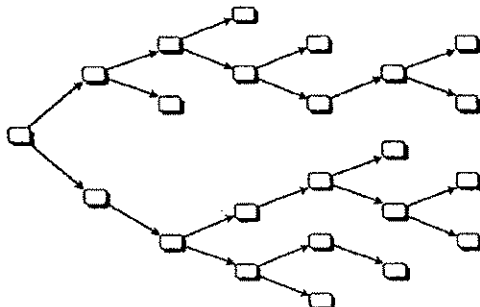


Figure 1.2: A more extensive view of the same worm.

1.1.2 Graph Building

In the worm example of the last section, we detected the worm based solely on the tree-like structure of the graph, and the nodes and edges of the graph were included based purely on time coincidence. GrIDS will use other information to determine if network activities are related, such as whether the traffic used the same destination ports or went to hosts of similar operating systems. In addition to forming graphs from network traffic reports, GrIDS will allow the nodes and edges to be annotated with *attributes*. These provide supplementary information about the node (host) or edge (traffic) in question. Hence, GrIDS will be able to detect worms based on events at the hosts and properties of the traffic, in addition to the shape of the graph. For example, if GrIDS were looking for a particular kind of worm, one that transmitted password files between hosts, then a worm-shaped graph would not be reported until one or more of the links were annotated as transmitting a password file.

Node and edge attributes may come from other IDSs, network sniffers, or any monitor that is equipped with a filter to send its output to GrIDS. A well defined syntax for reporting to GrIDS will be available to GrIDS users who wish to write their own filters. The GrIDS mechanisms make no specific assumptions about the nature of attributes, and GrIDS will be able to import externally written correlation functions. This will allow users to put the general GrIDS mechanism to work in many different ways. For example, thumbprinting [?]

mechanisms could be straightforwardly added in this manner. This would allow GrIDS to trace intruders through multiple connections.

Because GrIDS will search for numerous types of network abuse, it will need to build numerous kinds of graphs. A single graph containing all network activity would be too awkward to analyze effectively. GrIDS will maintain multiple *graph spaces*, where each graph space contains a number of graphs of a single type. The "type" of graph represented is determined by a set of rules which specify how graphs in that space are built. Each graph space has its own such *rule set*. A rule set only modifies the graphs in its own space, and has no effect in other graph spaces.

When new node or link information is received by GrIDS, the information is presented to each rule set, which determines whether and how the information is incorporated into graphs in the corresponding space. If the information is relevant to that rule set, it may be added to one or more existing graphs, cause multiple existing graphs to coalesce into a single graph, or be used to create a new graph. After such a change occurs in a graph space, the graphs in that space are analyzed according to certain rules in the rule set to determine if they are suspicious. If so, predetermined actions are taken, such as reporting the graph to the SSO. GrIDS will provide an interface for displaying such alerts.

GrIDS will have integrated capabilities to debug and trace rule sets selectively to help the user understand the operations of his graph rules. The user will be able to profile rule sets to determine how much time each takes to execute. This capability might show that certain rules are too expensive to operate; they should either be optimized or discarded.

1.1.3 Aggregation

GrIDS will model an organization as a hierarchy of departments. In order to manage this, it will provide a drag and drop interface which allows the hierarchy to be reconfigured during operation. Each department in the hierarchy has a GrIDS module of its own, which builds and evaluates graphs of activity within that department. However, activity which crosses departmental boundaries will be passed up to the next level in the hierarchy for resolution. That level will build *reduced graphs* in which the nodes are entire subdepartments rather than single hosts. More complex features of the full graph can be preserved in the reduced graph by use of attributes.

For example, consider the graph in Figure 1.3, which crosses several departmental boundaries. This graph might well be cause for suspicion taken as a whole. However, the graph is not particularly suspicious in any one department.

At the next level in the departmental hierarchy, the reduced graph (shown in Figure 1.4) will be seen. This graph is not suspicious just by nature of its topology. However, attributes of the individual subgraphs are passed up which allow the higher level module to draw stronger conclusions about the graph. For a simple example, each sub-department passes up

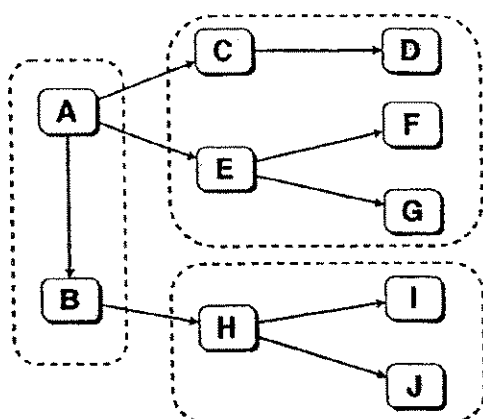


Figure 1.3: A graph amongst several departments. The dashed lines are departmental boundaries.

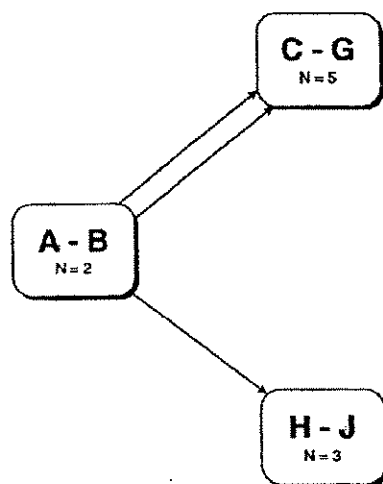


Figure 1.4: The corresponding reduced graph.

the size of the subgraph it sees. Thus GrIDS can deduce that the total graph has ten hosts in it. Similarly, the number of branches and the depth of the graph can be computed. This will be enough to render the graph suspicious.

Using this approach, GrIDS will be able to infer the suspicious nature of large graphs, while still reducing drastically the amount of information which must be considered at the top of the hierarchy. It is this that makes GrIDS a scalable design.

We also use the hierarchy to manage the rule sets. A rule set is specified at a particular node, and all the descendants of that node must implement that rule set. Other parts of the hierarchy will not share it. This eliminates potential ambiguity between similar rules in different parts of the network.

1.1.4 Policy

GrIDS includes a policy language to express unacceptable uses of computer networks. Policies are compiled into rule sets which watch for network activity that violates the policy. Hence, network abuses may be detected as the violation of a user specified policy, rather than a user specified rule set. Policies need not be written only for network abuses, though. An organization could include a policy to prevent users from accessing certain newsgroups during working hours, for example. The motivation for including a policy language in GrIDS is that it saves the user from having to write rule sets manually. In general, and even in the simple example below, rule sets are more complicated to specify correctly than is a policy. With a policy compiler, the user is brought one level higher in the abstraction.

As an example, let us consider the simple policy "no more than fifteen connections by a user at a time." This policy could be compiled to create a graph space where graphs are formed from connections labelled by user. The rule set would specify that each new connection be added to an existing graph made of connections by the same user, or create a new graph if none exists. Hence, the graph space would contain one graph per active user, and a violation would occur if any graph contained more than 15 connections. The rule set would, of course, need to specify how to analyze the graphs and report violations; this will be developed automatically from the policy.

We have analyzed a number of policies and shown that they can be expressed in our graph rule syntax.

1.1.5 Limitations

GrIDS tackles some of the hard issues which need to be faced for an intrusion detection system to operate on a large network. A lot of our effort is going into making the aggregation mechanism scalable, and allowing the system to be dynamically configurable so that it is still manageable when deployed on a large scale.

Nonetheless, we should be clear that the current version of GrIDS is intended as a proof of concept rather than as a finished system appropriate for immediate deployment in mission-critical environments. Hence, we do not ensure the integrity of communications between GrIDS modules, nor is anything done to prevent an attacker from replacing parts of GrIDS with malicious software of her own. The prototype will not be resistant to denial of service attacks, disruptions of the network time protocol, or faults in the networks or computers on which it runs.

The limitations above could be straightforwardly addressed in future versions of the system. A more fundamental limitation of GrIDS is that it is oriented towards detecting large scale attacks or violations of an explicit policy. It may not detect intrusions which are small, slow, or both. If other, local, IDS systems are installed, GrIDS can be used to present and manage their conclusions, but it will not detect all intrusions

itself. We like to think of GrIDS as radar. It is still possible for intruders to fly under the radar; however, it makes the intruder's task that much more difficult.

Chapter 2

The Graph Building Engine

2.1 Introduction

This chapter describes the GrIDS graph engine. This is the piece of software which takes basic activity reports and converts them into graphs. We first cover the purpose of the engine, then discuss its operation in detail.

2.1.1 Goals

The overall purposes of the graph engine are to build graphs of network activity and analyze them to see if they are suspicious. More specifically, the goals which we wish to achieve by building these graphs include

- To detect worms and sweeps.
- To detect network access policy violations.
- To trace the source of activities across the network.
- To add context to point detection reports.
- To provide the SSO with a comprehensible way to visualize the activities of particular users or hosts.

2.1.2 Overview

The graph engine is perhaps the most central piece of GrIDS. Its task is to take in reports of network activity, build graphs out of them, and report those graphs to other graph engines. There is a hierarchy of such engines (the administration of the hierarchy is described in more detail in chapter 6).

The graph engine has six inputs:

- Rules which describe how graphs should be built (section 2.3).
- Functions which can be loaded into the engine at runtime and then referred to in the rules (section 2.3.6).
- Reports of network activity and partial graphs in our graph description language (chapter 3).
- Messages from the organizational hierarchy server which instruct the engine on who it should report to, who is in its department, *etc.* (see chapter 6).

- Queries to see graphs matching a particular pattern (section 2.7).

- Messages controlling the nature of debugging information which is recorded (section 2.8).

It has four outputs:

- Reduced graphs of activity it has seen in its department (section 2.4).
- Full graphs of activity supplied in response to a query (section 2.7).
- Alerts sent to a user interface reporting suspicious graphs (section 2.6).
- Debugging logs (section 2.8).

The algorithms used by the engine are described in section 2.3.2, and then in more detail in section 2.10. Also relevant is section 2.5 which describes initial sorting and consolidation of incoming reports.

2.2 Graphs

Graphs consist of nodes, edges, attributes of nodes, attributes of edges, and global attributes of the graph as a whole. All attributes consists of a pair (*name,value*). Names are identifiers, while attribute values may either be scalars (in the Perl sense of being interchangeably strings, numbers, or logical values), sets of scalars, or ordered lists of scalars.

Reports to the graph engine are in the form of (parts of) graphs, and the output from the graph engine is also in the form of a graph. The language in which graphs are specified is detailed in Chapter 3.

2.3 Graph Engine Rules

The graph engine takes as input a set of rules which tell it how to build and analyze graphs. This section discusses the syntax and semantics of those rules.

2.3.1 Rulesets

Not all types of network activity are related enough to belong in the same graph. Independent graphs representing different kinds of behavior may be easier to analyze than a single graph containing several largely unrelated kinds of information.

For example, one user may wish to look for many telnet connections being generated from a single host, while another wishes to look for a series of rlogins from host to host to host. If both kinds of connections (and perhaps other kinds as well) are all contained in one graph, it will be difficult to detect the relevant patterns. If, however, one graph contains only telnet connections, and one contains only rlogin connections, then the users merely need to look at the shape of their graphs to determine if their criteria have been met (because they know all edges in their graphs are of the type they are interested in.)

For this reason, the graph engine is capable of maintaining multiple graph spaces, where each graph space contains only graphs of one type. The "type" of graph is specified by the rules which dictate how graphs in the graph space are built. In the example above, one graph space would have rules that allowed only telnet connections to be added to its graphs, where another graph space would allow only rlogin connections to enter its graphs.

The rules in different rule sets do not interact at all. Each operates on its own graphs and does not affect the graphs of other rulesets.

2.3.2 An Overview of How the Rules Work

Reports come in as partial graphs. For each ruleset, the graph engine maintains a set of existing graphs. In processing an incoming report, the task of the engine is to determine which graphs the incoming report should be combined with and how the attributes should be updated. The rules are used in several ways.

Firstly, there are *preconditions* for the ruleset. These determine whether the incoming report is of a type that is suitable to be incorporated into this ruleset at all. If not, that is the end of the application of the rules for this particular ruleset for the report. There are two kinds of preconditions, *node preconditions* which are applied at every node of the incoming report, and *edge preconditions* which are applied at every edge. If any node or edge passes the precondition, then the report as a whole is deemed to pass the precondition.

Provided the precondition is met, the partial graph gets developed into a full graph, with all the attributes associated thereof. How this transformation takes place is indicated by *report rules*, which dictate how the attributes of the graph, nodes, and edges should be created based on the attributes in the partial graph. Now that a full graph has been made from a report, the next step is to possibly combine this new graph with its peer graphs in the graph space.

An adjacency test is now applied between the new graph and each of the other graphs constructed by the ruleset. The

incoming graph will only be considered for incorporation into graphs with which it has an overlap in at least one node or edge. It will not be incorporated into graphs with which it is wholly disjoint.

Next, there are conditions, which are used to determine whether an incoming graph should be combined with some particular existing graph with which it does have some overlap. If the report is an aggregated graph attribute report from a subdepartments engine, then the graphs with which to combine are determined as described in section 2.4. Otherwise rules in the ruleset are used. These rules are again specific to nodes or edges. They are applied on every node or edge of the incoming report. Assuming that at least one of these conditions evaluates to true, then the incoming report will be incorporated into that existing graph. If no edge or node condition is true, then the report will not be incorporated.

It may well be that the new graph can be incorporated with several incoming graphs. If so, then the report will automatically trigger those graphs and the new graph to combine in a pairwise manner. A set of rules are concerned with how to combine two graphs. The actual combining of nodes and edges can be handled automatically, but the recomputation of attributes is something which the user must specify; the engine does not know how to compute the user's attributes. Firstly, there are rules which dictate how to combine global attributes of the two graphs. Then there are rules which depend on local attributes at the nodes and edges in the intersection of the graphs. The local rules may make modifications to the global attributes also, to account for global attributes which depend on the graph topology and which cannot be computed purely from global attributes of the combining graphs.

2.3.3 Graph Rules - A Simple Example

An example should serve to make the rule operation clearer. The rules consist of a series of rulesets one after another. We take the example of a ruleset which is intended just to detect worms by aggregating connections together which occur close together in time. It also includes any node reports which have an *alert* attribute if they fall in the appropriate time frame.

The first line of a ruleset simply specifies that the text of a new ruleset is beginning and what its name is. The second specifies what length of α -buffer to use for this particular ruleset. Here we have no buffer. The timeout line indicates how long to wait since the last addition to the graph before delete the graph (provided no new additions are made).

```
ruleset worm_detector;
```

```
buffer 0;
```

```
timeout 600;
```

Next we specify some macros (some text to replace by some other text), which in this case are just constants.

```
macros { L=30; }
```

Next, there are report attribute declarations. These specify the nature of any attributes which are going to be referred to in the rules. In this case there will be *time* and *alert* attributes which are scalars. Any attributes which are not specified in the ruleset, but show up in reports anyway, will simply be ignored.

```
attribute declarations {
time scalar;
alert scalar;
}
```

A report arrives in the form of part of a graph. The first thing that happens is that, for each ruleset, the report is examined to determine if it is appropriate to incorporate in that ruleset. For each node and edge in the incoming graph, an appropriate node or edge precondition is evaluated. If the precondition evaluates to true for *any* such node or edge, then the report will be processed further against this ruleset.

In this case, the node precondition looks like the following:

```
node precondition defined(new.node.time) &&
                    defined(new.node.alert);
```

The expression `defined(new.node.time) && defined(new.node.alert)` says that the incoming node must have a *time* attribute and an *alert* attribute in order for this condition to be true. Recall that Grids scalars are derived from those of Perl – they evaluate to true if they exist and are non-zero or non-empty strings and have an explicit “undefined” value if no value has been set. Thus an isolated node report will only be considered for incorporation into the graphs of this ruleset if it has both of these attributes defined. The new syntax specifies that it is the attribute of the incoming node that is under consideration. In this case, there is no ambiguity (in fact the “new.” can be omitted), but that will not be true as we move along.

The precondition for edges to be considered is more lenient; they simply have to have a time attribute:

```
edge precondition defined(new.edge.time);
```

So, if an incoming report has any node or edge which satisfies the corresponding condition, then that report will be incorporated into the graphs for the ruleset in some manner, as determined by later rules.

First, however, reports must be made into graphs. The nodes and edges in the incoming reports appear in the resulting graph, but the attributes (except the auto-computed ones, see section 2.3.4) must be computed explicitly in the following rules. In these rules, *new* refers to attributes appearing on the report and *res* refers to attributes being computed for the graph, which is the default.

```
report global rules {
res.global.alerts = {};
res.global.time = 0;
}
```

The “report global rules” are run first for a first cut at computing the global graph attributes. In this example, the *alerts* and *time* global attributes are initialized.

```
report node rules {
res.global.alerts = {res.global.alerts,
new.node.alert};
res.node.alerts = {new.node.alert};
res.global.time = max({res.global.time,
new.node.time});
res.node.time = new.node.time;
}
```

Next the node rules for making a report into a graph are run for each of the nodes in the incoming report in some order. This section has access to both the set of global attributes and the attributes on the report node and update the “res” global and the attributes on the corresponding node. The above rules combine any *alert* attributes of the node with the global *alerts* attribute, putting the result into the the global *alerts* attribute; initialize the local *alerts* attribute; and does something like what was done for the *alerts* attribute for the *time* attribute, except that the maximum is kept around.

```
report edge rules {
res.global.alerts = {res.global.alerts,
new.edge.alert};
res.edge.alerts = {res.edge.alerts,
new.edge.alert};
res.global.time = max({res.global.time,
new.edge.time,new.source.time,new.dest.time});
res.edge.time = max({new.edge.time,
new.source.time,new.dest.time});
}
```

These edge rules are like the above node rules except that they are executed for each edge rather than each node.

There are a number of complications to understand about the next parts of the rules. Two graphs are being considered for combination, and also the way that the attributes on two graphs will combine into a final graph is being specified. Graphs combine if they overlap in at least one node or edge and if the combine rules evaluate to true for any node or edge in the intersection of the graphs. Computations are only done with the global attributes, and with the nodes and edges that are in the overlap of two graphs.

The following three rule sections describe whether and how to combine two adjacent graphs and are similar in structure to report rules above. In the following rules, *res* will refer to attributes being computed, while *new* refers to attributes appearing on one of the graphs, and *cur* refers to attributes

on the other graph. If none of *res*, *new*, or *cur* is mentioned, then the resulting graph is the one being referred to.

The first rules to appear specify how the global attributes of two graphs should combine. This will be a first effort at combination which can be modified by local rules further down.

```
global rules {
  res.global.alerts = {new.global.alerts,
    cur.global.alerts};
}
```

This says that the final global *alerts* attribute for a graph will be the union of the existing *alerts* attribute for one graph, together with any global *alerts* attribute in the other graph. The notation { ... } is a set constructor, where sets inside the constructor are flattened out. Set and list constructors are described in more detail in section 2.3.5.

Next we have the rules which specify how nodes combine. The order in which node and edge rules are evaluated is implementation dependent and should not be relied upon in writing rules.

```
node rules {
  res.node.combine =
    !empty({new.node.alerts, cur.node.alerts}) &&
    abs(cur.node.time - new.node.time) < L;
  res.node.alerts = {cur.node.alerts,
    new.node.alerts};
  res.node.time = max({cur.node.time,
    new.node.time});
}
```

The first rule shown concerns a special fictional attribute *combine*. The value of this attribute will not appear in the final graph, but rather is used in the computation of whether the graphs should be combined at all. If the *combine* attribute evaluates to true on any overlapping node or edge in the graphs, then the graphs will in fact be coalesced. In the particular case above, the condition for the graphs to combine based on a particular node is basically that one of the nodes has a non-empty *alerts* attributes, and that the time attribute on the one node and the time attribute on the other node are sufficiently close together (where sufficiently is here defined by *L* which was earlier set to 30 seconds in a macro – all GrIDS times are in seconds).

The remaining node rules specify, assuming that the graphs are to combine at all, how attributes at nodes will combine. For example, the *alerts* attribute at a node in the final graph will be the union of the *alerts* attributes for the nodes and the *time* attribute will be the latest of the *time* attribute on the nodes. Not that “res.global” attributes could have been updated in these rules if the rule writer so desired.

Now, we have a set of edge rules which have a similar function as the node rules.

```
edge rules {
  res.edge.combine = abs(source.cur.time
    - new.edge.time) < L;
```

This first edge rule is the one that dictates whether combination should be triggered by this rule or not. In this case, the condition applied is that the *time* attribute on the incoming edge must be sufficiently close to the *time* attribute on the node which is the source of this particular edge in the existing graph.

The remaining edge rules determine the value of local and global attributes. These are all similar in nature to the node attribute rules described above.

```
res.edge.alerts = {cur.edge.alerts,
  new.edge.alerts};
res.edge.time =
  max({cur.edge.time, new.edge.time});
}
```

Finally, we have the assessment rules which evaluate the resulting graph and take appropriate actions. The resulting graph attributes may be referred to as “res” or without such a prefix. Assessment rules may only refer to global attributes. The actions on the right hand side can either be calls to built-in or user defined functions or assignments to global attributes. The alert function sends some text as an alert to the user interfaces monitoring the engine and the report_graph function sends some text and the current graph to the user interfaces. The numeric argument to these functions is an alert level. This is described in more detail in section 2.6.

```
assessments {
  !empty(global.alerts) ==>
    report_graph(2, "alerts found!");
  global.nnodes > 7 ==> report_graph(3,
    global.nnodes::" nodes in graph");
  global.nedges > 12 ==> report_graph(2,
    global.nedges::" edges in graph");
  global.nnodes > 3 ==>
    alert(1, "warning from "::global.ruleset);
  global.nedges > 5 ==>
    alert(1, "> 5 edges in graph");
}
```

The point worthy of note here is that a number of attributes are being referred to which the earlier rules did not compute. These are automatically computed attributes, which can be referred to by the rules, but not assigned to. More detail on these attributes can be found in section 2.3.4.

2.3.4 Auto-Computed Attributes

The following attributes are computed automatically by the engine. They may be referred to by the rule writer, but are not computed explicitly by the rules.

- Global Attributes

- *gids*, a set of graph ids associated with this graph, any of which can be used as a unique identifier.
- *ruleset*, the name of the ruleset this graph is in.
- *nnodes*, the total number of nodes in a graph.
- *nedges*, the total number of edges in a graph.
- Node Attributes
 - *name*, the name of this particular node (see chapter 3).
- Edge Attributes
 - *source*, the domains associated with the source of this edge that are within this aggregator's domain, in a list starting with the domain for the source within this aggregator's domain and ending with the host (see section 2.4).
 - *dest*, same as *source* except pertaining to the destination side of the edge.
 - *id*, a textual unique identifier for this edge.

2.3.5 Rule Primitives

The operations available in the engine rules are described in this section. The following binary infix operations are available:

- *+*, integer and floating point addition
- *–*, integer and floating point subtraction
- ***, integer and floating point multiplication
- */*, floating point division
- ****, floating point exponentiation
- *::*, string concatenation
- *==*, numeric equality
- *<*, numeric less than
- *>*, numeric greater than
- *<=*, numeric less than or equal to
- *>=*, numeric greater than or equal to
- *eq*, string equality
- *lt*, string less than
- *gt*, string greater than
- *ne*, string inequality
- *&&*, boolean short-circuit and
- *||*, boolean short-circuit or

- *::*, string concatenation

One unary operation is available, *!* which is the boolean "not" operation.

Set and list constructors are also available. The set construction begins with a '*{*' and ends with a '*}*' and contains a comma-separated list of scalars, sets and lists. All the scalars (including those in the sets and lists) become the elements of the constructed set. For example, *{ "a", { "b", "c" } }* evaluates to the set containing "a", "b", and "c". This list constructor is similar, but is surrounded by '*[*' and '*]*'.

The following primitive operations are available using function-call syntax:

- *in_set(scalar,set)*, evaluates to true if the first argument is contained in the set given in the second argument, false otherwise, where string comparison is used
- *max(set)*, evaluates to the numerically largest scalar in the set
- *min(set)*, evaluates to the numerically smallest scalar in the set
- *empty(set)*, evaluates to true if and only if the given set is empty
- *on_list(scalar,list)*, evaluates to true if the first argument is located somewhere on the list given in the second argument, false otherwise, where string comparison is used
- *head(list)*, evaluates to the first scalar on the list
- *last(list)*, evaluates to the last scalar on the list
- *sort(list)*, evaluates to a list containing the same scalars as the given list, but in lexically sorted order
- *sort_numerically(list)*, evaluates to a list containing the same scalars as the given list, but in numerically sorted order
- *abs(scalar)*, evaluates to the absolute value of the given number
- *alert(alert_level,message)*, sends an alert with the given level, message, and no graph (see section ?? on alerts)
- *report_graph(alert_level,message)*, as with *alert* but sends the current graph as well

2.3.6 Importing Functions into the Graph Rules

The graph rules allow user supplied functions to be used in the rules in addition to the built in functions provided by GrIDS. This section describes the syntax and calling convention for such functions.

Three functions must be written in Perl. The arguments supplied to the user defined function will be either GrIDS

scalars, sets of scalars, or lists of scalars. The user must supply a function prototype and a perl implementation of the function. The prototype is of the form:

```
list foo(set, scalar, scalar, list, ...)
```

In addition to scalar, list, and set, "void" is a valid return type from a function, indicating that no return value should be expected. Providing prototypes enables GrIDS to check the types of invocations of this function in the rules.

The conventions on arguments to and return values from the function are as follows:

- GrIDS scalars become Perl scalars.
- GrIDS lists become references to Perl lists.
- GrIDS sets become references to Perl associative arrays. Elements in the GrIDS set hash to a true value, while elements not in the set hash to 0 or are not present in the associative array.

The functions and prototypes are supplied to the engine in one or more files. Each file has two parts:

- a prototype part, beginning with the start of the file and ending with the start of the code section (or the end of file), this section consists of some lines with prototypes and some lines which are ignored. If a line begins with "void", "set", "scalar", or "list", possibly preceded by a "#", then the line is taken as a prototype.
- a code part, starting with the first line of the file that starts with a "sub", the section consists of a set of perl functions which implement the supplied functions. A function begins with a line starting with a sub and ends with the line before a line starting with a "sub" (or the end of file).

The prototype of a function must precede its implementation, in whatever order the files are read in.

2.3.7 Rule Grammar

The grammar for the rule language is specified here.

```
<rules> => <ruleset>*
```

```
<ruleset> => 'ruleset' <id> ';'
           'buffer' <constant> ';'
           'timeout' <constant> ';'
           <macros> <declarations>
           <preconditions> <report-rules>
           <combine-rules> <assessments>
```

```
<macros> => macros '{' <macrolist> '}'
```

```
<macrolist> => <macro> <macrolist>|λ
```

```
<macro> => <id> '=' <constant> ';'
```

```
<constant> => [0-9]+ | [0-9]+ '.' [0-9]+ |
               '\'' ['\"']* '\'
```

```
<declarations> => 'attribute declarations'
                 '{' <decl-list> '}'
```

```
<decl-list> => <declaration>;<decl-list>|λ
```

```
<declaration> => <id> 'set' | <id> 'list' |
                 <id> 'scalar'
```

```
<preconditions> => <node-precondition>
                  <edge-precondition>
```

```
<node-precondition> => 'node precondition'
                      <expr> ';'
```

```
<edge-precondition> => 'edge precondition'
                      <expr> ';'
```

```
<report-rules> => <report-global>
                 <report-node> <report-edge>
```

```
<report-global> => 'report global rules'
                  '{' <assignment>* '}'
```

```
<report-node> => 'report node rules'
                '{' <assignment>* '}'
```

```
<report-edge> => 'report edge rules'
                '{' <assignment>* '}'
```

```
<combine-rules> => <global-rules> <node-rules>
                  <edge-rules>
```

```
<global-rules> => 'global rules'
                  '{' <assignment>* '}'
```

```
<node-rules> => 'node rules'
                '{' <assignment>* '}'
```

```
<edge-rules> => 'edge rules'
                '{' <assignment>* '}'
```

```
<assignment> => <attribute> '=' <expr> ';'
```

```
<assessments> => 'assessments'
                 '{' <assessment>* '}'
```

```
<assessment> => <expr> '=>' <actions> ';'
```

```
<actions> => (<action> ',')* <action> | λ
```

```
<action> => <assignment> | <function-call>
```

```

<attribute> => <object-indic> '.' <id> | <id>

<object-indic> => <graph-indic> <domain-indic>

<graph-indic> => 'res.' | 'cur.' | 'new.' | λ

<domain-indic> => 'global' | 'node' | 'edge' |
    'source' | 'dest'

<expr> => <attribute> | <constant> | !<expr>
    <expr> <binary-op> <expr> |
    <function-call> | '(' <expr> ')' |
    '{' <expr-list> '}' |
    '[' <expr-list> ']' | <regex>

<regex> => <attribute> '=' '/' <anything>* '/';

<expr-list> => <expr> | <expr> ',' <expr-list> | λ

<binary-op> => '+' | '-' | '*' | '/' | '**' | '=' |
    ':' | '&' | '|' | 'eq' | 'ne' | 'gt' | 'lt' |
    '<' | '>' | '<=' | '>=' | '<' | '>'

<id> => [A-Za-z_][A-Za-z0-9_]*

<function-call> => <id> '(' <expr-list> ')'
```

The above grammar is more permissive than the semantics actually allow. For example, 'source' and 'dest' only have meaning within edge rules although the grammar allows it elsewhere as well. Node attributes should only be assigned to inside node rules and edge attributes only within edge rules.

2.4 Aggregation of Graphs

2.4.1 Introduction to Aggregation

Graph engines build graphs of activities within a certain portion of a network. Take as an example, the graph engine which collects reports from all the machines within the Production department of ACME inc. These machines do not necessarily fit physically on a single network, but from a human perspective, they all lie within the Production department.

Often, two machines within the Production department may connect, as shown in Figure 2.1. The report of the connection is seen by the engine in the production department.

Sometimes, a connection from within the department of a particular graph engine is made to a host outside that graph engine's realm. For example, in Figure 2.2, a connection within the Production department is made to a host in the Management department. The management end of the connection is clearly outside the realm of the Production department's graph engine, yet the connection is visible to it. The Production department's engine builds a graph as shown in Figure 2.3, but the engine must also "pass up" to

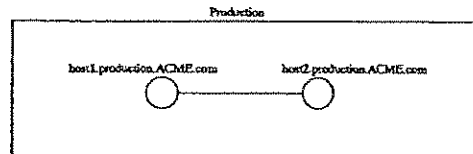


Figure 2.1: Two machines connecting in the production department.

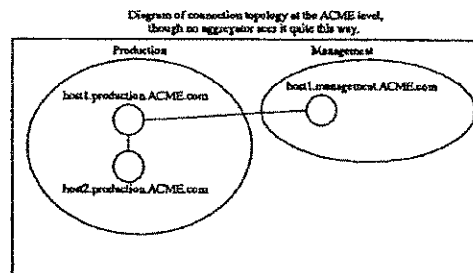


Figure 2.2: Connection from production to management.

a higher level engine (i.e., aggregator) this report that goes outside its realm. To do this, engines must be able to recognize nodes that are within their department and nodes that are not. Hence, engines keep record of all host and department names which lie within their realm. In this example, the engine in the Production department sends a report to the aggregator for ACME inc. because it does not recognize "host1.management.ACME.com" as a department or host that Production contains.

We assume that an engine for ACME exists and is running to accept this report. It receives a report from Production and sees that the connection occurs between two departments both of which lie within ACME. (Similarly, the Management department will have built its own graph and sent a report up to ACME's engine.) The ACME Inc. engine creates the graph shown in Figure 2.4, showing the two departments as nodes and the hosts within those departments that were involved as attributes on the link.

Imagine now that the entire .com domain is watched by a GrIDS engine. When a connection comes out of ACME Inc

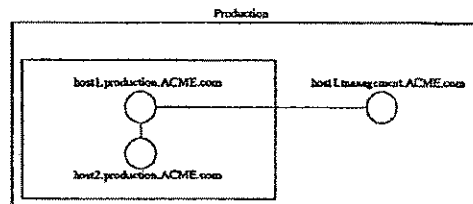


Figure 2.3: The production department graph engine's view of the same situation.

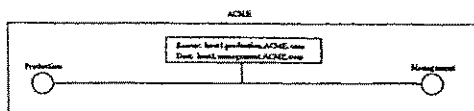


Figure 2.4: The view according to the engine for Acme Inc.

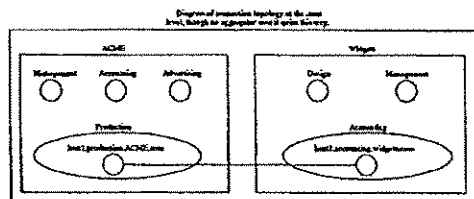


Figure 2.5: Connection between ACME and Widgets.

to Widgets Inc, as shown in Figure 2.5, the ACME aggregator must "pass up" news of this to the .com engine, which builds the graph shown in Figure 2.6. As in Figure 2.4, the link is decorated with the names of the hosts involved in the connection. But this time, there were also departments involved which do not show up at the engine's level. Hence, they too are denoted on the link. In general, all departments and hosts on the source and dest side that cannot be seen at the engine's level will be listed on the link, recursively.

More realistically, imagine that ACME is running GrIDS and Widgets is also, but no overall .com GrIDS engine exists. Also imagine that ACME and Widgets do not share GrIDS data. (Perhaps they don't trust each other.) When ACME's engine sees a connection going out to Widgets, it produces a graph as shown in Figure 2.7. The graph shows that a connection comes out of host1 within the Production department (both of which are known) and goes to some unknown machine called host1.accounting.widgets.com. Because, in this example, there is no GrIDS engine running at a higher level than ACME, the ACME engine does not aggregate the graph shown in Figure 2.7 upwards.

2.4.2 Aggregating Attributes

Links in graphs show all hidden departments and hosts involved in the link. This is done through the use of the automatically computed "source" and "dest" link attributes. Having this information available allows the rules to analyze the links more fully and make more precise decisions regarding them. For the same purpose, at-

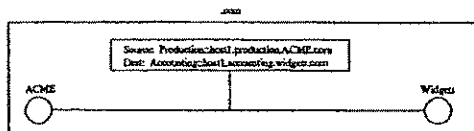


Figure 2.6: View at .com domain

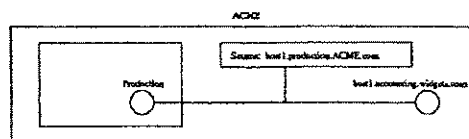


Figure 2.7: ACME view of connection to Widgets.

tributes of hidden departments and hosts may be included in the link information, but only through explicit storage of such attributes. Look again at Figure 2.5. When the connection between host1.production.ACME.com and host1.accounting.widgets.com is seen by the .com engine (assuming again that it exists), the node attributes of host1.production.ACME.com and host1.accounting.widgets.com would not necessarily be available, though they may be of interest.

2.4.3 When Reports Are Sent to the Parent Engine

Whenever a graph engine makes a change to the global attributes of a rule set, the new attributes for that space is passed upwards (seen at the next level up as a node report.) Whenever a connection is entered to a graph space which contains one side in the realm of the engine and one side out of the realm of the engine, a report of the connection is passed upwards with the attributes present on the edge after the execution of the rules. Both of these upward propagations are on a per-ruleset basis, with the reports from a particular ruleset only being applied to the same ruleset at the higher level. Note that the realm of engine at the next level up may still not contain the foreign host, in which case it can enter the link into its graphs but then must pass it up as well.

2.4.4 Graph Aggregation Details

Recall that a department shows up as a node in the parent domain's aggregator and that rulesets may permit multiple instances of nodes with the same name to appear in separate graphs within a ruleset. Not only are graph attribute reports destined for a particular ruleset in the parent, they are also, in general, destined for a particular instance of a node.

Associated with each graph being constructed by the engine is a graph identifier (gid) that is unique within the ruleset for a department. Similarly, associated with each node that represents a department is an instance identifier. The instance id on a node is the same as the gid of the graph whose global attributes correspond to the node's attributes. Thus there is a correspondence between a graph at a lower level and a node at a higher level. When the global attributes on a graph change, the updated attribute values are sent to the particular node in the parent that corresponds to the graph at lower department level (i.e. has the same name as that of the department and has the instance id that is that same

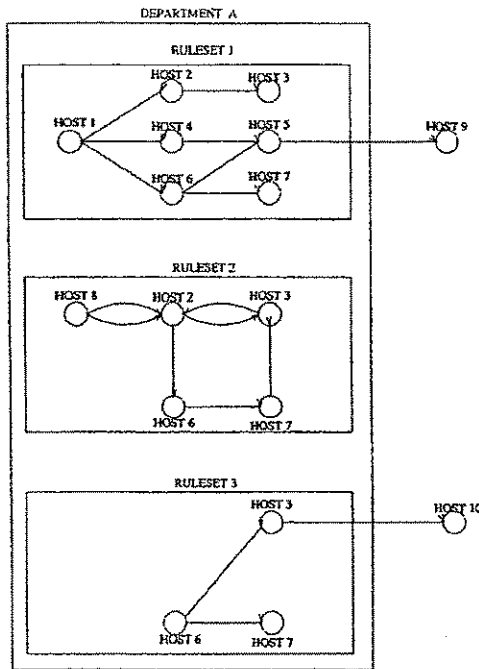


Figure 2.8: Graph Spaces At Host Level

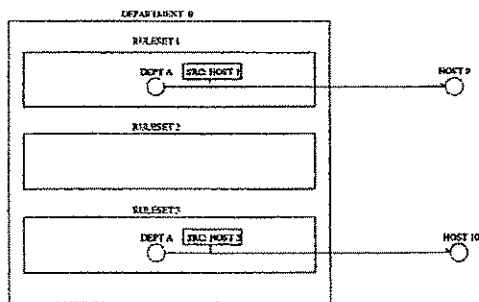


Figure 2.9: Graph Spaces At Aggregated Level

as the gid). The attributes that are sent up are constructed into a graph (as per the rules in the ruleset) and merge (also as per ruleset rules) with the graph containing that instance, thereby (potentially) updating the attributes on a node and graph.

However, as time proceeds, graphs can be combined at the lower level, the graphs at the higher level can merge (thus causing the department node instances to merge), and graphs at either level may be inactive long enough to time out. There is also the case of the first attribute report from a graph and the case where the attribute report was rejected at the higher level. These cases must be dealt with since there is no longer a one-to-one correspondence between graphs and node instances.

The way this is done is that when two graphs combine, the resulting graph inherits the gids of each graph. Thus as time goes along and graphs merge, the set of gids on a graph include all the gids of all the graphs that contributed to the forming of that graph. When an attribute report is sent up, it is sent to each of the node instances corresponding to the gids on the graph. The graphs containing the instances are used as a more restricted set of graphs to potentially combine with than in the non-aggregated case (where any graphs in the ruleset might combine, assuming they pass the adjacency test). If the graphs containing those instances decide to accept the report (by merging with it), the usual thing happens when reports are accepted into multiple graphs, that being that all the graphs are combined. In the case where there are no graphs containing any of the node instances to which the report was sent (perhaps they all timed out), then the aggregated report is treated like a normal (non-aggregated) report and may be combined with any graph in the ruleset.

When instances of a node combine (because the graphs containing the combined), the instance ids then correspond to the same node as would be expected. In a sense, the instance ids are inherited in much the same manner as gids were inherited when graphs combined. In the case where graph reports from a graph do not get incorporated into the graph that the last report from the same graph did (due to restrictive combining rules), the new graph formed will be the target of the next attribute report from that graph. It should be noted here that not all nodes have instance ids associated with them, just those that correspond to departments (as opposed to hosts).

2.4.5 Graph Aggregation Example

An example should help make things clearer. First let's define some notation. Let a subscripted name of a department or host that is capitalized denote a graph in that department and one that is subscripted and in lower case denote a node in the parent's engine that corresponds to the department. The subscripts on each of these are the id(s) associated with the graph or node.

Consider a department A which is one of Department B's children. At some point there is a graph in A, A_1 , which corresponds to a node a_1 in one of B's graphs, B_7 . Changes in

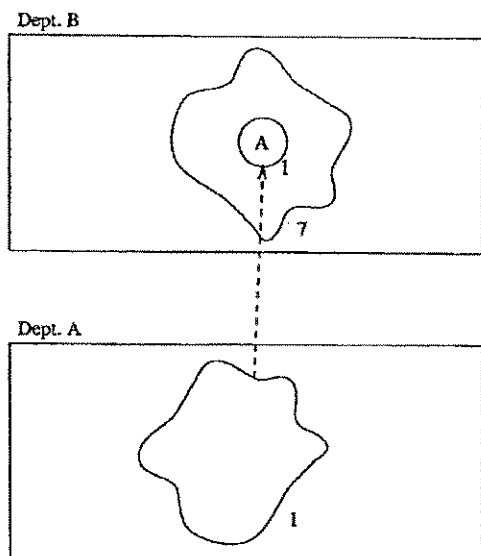


Figure 2.10: Example graph aggregation attribute path (time 1)

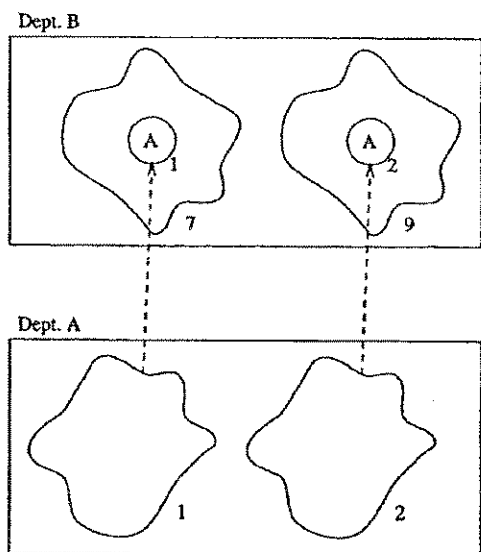


Figure 2.11: Example graph aggregation attribute path (time 2)

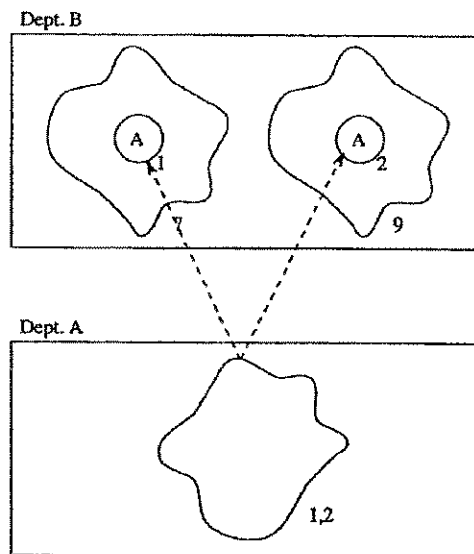


Figure 2.12: Example graph aggregation attribute path (time 3)

A_1 's global attributes are sent to B_7 since B_7 contains a_1 . (see Figure 2.10.) Now suppose a new graph A_2 is created. Suppose that its graph attributes propagated upward and caused a new graph B_9 to be created within B's engine. now changes in A_1 's global attributes are sent to B_7 and changes in A_2 's global attributes are sent to B_9 as shown in Figure 2.11.

After a fashion, A_1 and A_2 merge forming $A_{1,2}$. Now all attribute reports from $A_{1,2}$ are sent to both B_7 and B_9 . (see Figure 2.12.) Later on A_3 gets created and its initial attributes propagating upwards (which don't end up being directed at an particular graph) is intermediately put in newly created $B_{1,2}$ which contains a_3 . At the merge phase, $B_{1,2}$ and B_9 merge, forming $B_{9,12}$ which necessarily contains a node $a_{2,3}$. As Figure 2.13 shows, at this point changes to $A_{1,2}$'s global attributes are still sent to both B_7 and $B_{9,12}$ and changes to A_3 's attributes are sent to $B_{9,12}$. Thus note it is possible for multiple graphs in a lower level to correspond to the same higher level node.

2.5 Buffers

The GrIDS engine has three buffering mechanisms to serves different ends:

- to sorts events, which could come in out of order, by time,
- to remove redundant reports, and
- to adjust reports to be correct from a particular engine's point of view.

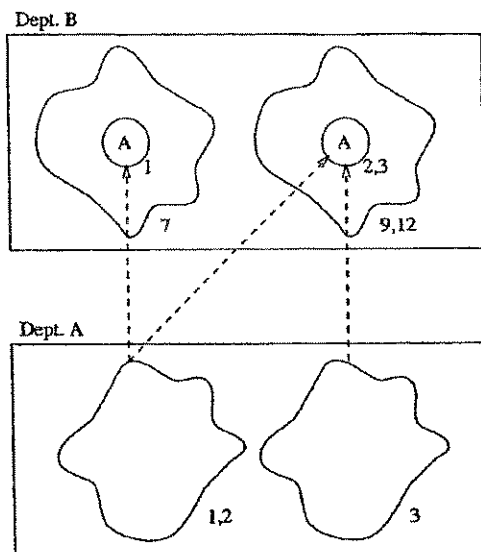


Figure 2.13: Example graph aggregation attribute path (time 4)

All incoming reports are assumed to have a *time* attribute which represents the true time at which the event to which they refer occurred.

The α buffer sorts them by this attribute as much as possible, before giving them to the rest of the engine. As part of any given ruleset, a lag time $\alpha \geq 0$ is specified. The buffer will correct any out of order events if they are not out by more than α . Note that differences between the system clocks on the source and engine hosts affects how well the buffer works. If an incoming event is already out of date by more than α on arrival, then it is handed off to the rest of the engine immediately.

The β buffer acts to coalesce multiple reports of the same thing. Since the buffer stores things in time order, it can check when an event is reported if it has been reported already in an identical manner (by another sniffer, for instance) by looking back in the buffer β time. If any entries match as being the same (a best guess, subject to engine implementation details), the "new" data is dropped. This is valuable for networks ripe with sniffers, where each link activity may be reported identically many times. If two data sources witness the same thing, but report it differently, both reports will go into the buffer to be fed to the rest of the engine.

It is beneficial to remove redundant reports since adding identical reports into the engine causes the engine to work multiple times. The β buffer reduces this by eliminating all but one of the reports. However, the buffer is only β long, so if some data source has enough lag time in reporting, this late report may not be eliminated. For efficiency, β should be set (according to a network's behavior) to a value that accounts for most lags, but does not make the buffer too long. The

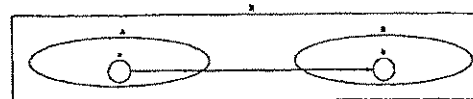


Figure 2.14: Connection between host a in department A and host b in department B.

individual rulesets do not specify β like they do α . Instead, it is specified for the graph engine as a control variable (see section 2.9). Hence, each ruleset can be thought of as having its own ordering buffer, but sharing a common redundancy reducing buffer.

The γ buffer serves to reconcile the hosts or departments mentioned in reports with the departments in which they belong for the perspective of a particular engine, which operates at a particular department. If a host or department is (recursively) contained in the engine's department, then its node is renamed to be the appropriate child of the engine. The *source* and *dest* auto-computed attributes are also updated. The reason this step is needed is shown in Figure 2.14. There, department A reports a connection from host a to host b as A to b whereas department B would report the same connection as being from a to B. From the point of view of department B, the report should be from A to B. The engine could have been designed to know all of its descendants, but this might require updating several engines every time a host or department changed departments. Instead, the gamma buffer monitors incoming reports for the *source* and *dest* attributes of edges to automatically discover its hierarchy. However, initially no information is known about the child corresponding to a reported host or department. This issue is addressed by buffering reports whose source and destination cannot be immediately resolved to be descendants of the department. They are kept in the buffer for up to γ seconds. The engine attempts to resolve the department for the nodes for up to γ seconds. If either the source or destination is a child of the department, then the report is assumed to be fully resolved and any remaining unresolved nodes are foreign to the department of the engine. Otherwise, the report is discarded as not of interest for the department, since both hosts sides are out of the scope of this department. Note that in the example above, provided that the two reports are given to the engine within γ seconds of each other, then both reports could be properly reconciled regardless of whether the engine knew that host a was in department A and that host b was in department B beforehand.

The order of the buffers for a new report is β , γ , then α .

2.6 Alerts

When a ruleset detects what it considers to be suspicious activity, it sends an alert message to the user interfaces that are monitoring the engine (as determined by the organizational hierarchy server, chapter 6). The rule primitives "alert" and

"report_graph" are used to send these alerts. See the rule primitives section (section 2.3.5) for details on these primitives.

Alerts always contain the name of the ruleset that generated it, a message, and an alert level. The message is some text to indicate the the user what the alert is about. The alert level is a number between 1 and 3 which the user interface uses to indicate the seriousness of the alert, with higher numbers indicating more serious alerts. Alerts may contain the graph corresponding to the generated alert in the standard GrIDS format described in section 3.2.

Alerts are sent using the GrIDS common packet format (chapter 3.1). The header used is "alt" and the body contains these elements in order: the ruleset name, the message, the graph (or the empty sting if no graph is being sent, and the alert level.

2.7 Query language

For increased usability, the graph engine accepts queries regarding its graphs. Queries select which graphs are to be returned.

English examples of permitted constraints follow: "Contains lhotse", "contains a link between lhotse and denali", "within ruleset1 containing lhotse with the attribute 'compromised'."

The engine checks each graph it contains against the list of statements. If all the statements are true for a graph, then that graph is returned.

For efficiency of checking, not all types of statements are allowed. Referring to attributes are on some node or edge is not allowed, because the engine would have to search through each node/edge to check for a match and this might slow the engine significantly if the engine contains many graphs of if the graphs are large. However, once you have specified the global attribute, or the attribute on a specific node/edge, you may perform operations on that attribute that evaluate to TRUE or FALSE. The statement is considered true if the operation evaluated to true. Specific nodes are identified by their "name" attribute. Edges are identified by their source and destination. Note that this may be ambiguous.

The syntax of the constraints is the same as that of individual ruleset rules except that attributes are specified with different prefixes. Bare attributes (i.e. "a") and attributes prefixed with "global" indicate global attributes of the graph. Those attributes of the form "<text>.node.<id>" refer to attributes of the node with the name equal to the indicated text. For example "lhotse.cs.ucdavis.edu".node.time refers to the "time" attribute on the node with attribute "name" equal to lhotse.cs.ucdavis.edu. "lhotse.cs.ucdavis.edu -> sierra.cs.ucdavis.edu".edge.time refers to an edge from "lhotse.cs.ucdavis.edu" to "sierra.cs.ucdavis.edu". If there is more than one edge, one is chosen in some unspecified manner.

Queries corresponding to the above examples are:

```
defined("lhotse".node);
```

```
defined("lhotse -> denali".edge);
defined("denali -> lhotse".edge);
```

```
ruleset eq "ruleset1" && defined("lhotse".node)
&& "lhotse".node.compromised;
```

GrIDS packets coming into the engine with the header "q" are interpreted as queries with the text of the query in the body.

The engine returns the query results to the querier in the GrIDS common packet format with the header "qr". The return value of a query is a composite graph of all graphs which passed the tests (see chapter 3.1). The following grammar is used for the list:

```
<graph-list> -> [<graph> <graph-list>]
```

where graph is defined in the communications protocol grammar.

2.8 Debugging

[This section is inconsistent and needs to be revised.]

2.8.1 Introduction To Log Files

When a graph engine is first brought online and starts receiving reports, it starts numbering the reports. (The numbers are unique across invocations. How this is done is indicated later.) All reports it receives are numbered roughly in order of when the graph engine received them. It isn't important to have them ordered exactly right, only that the number be unique for each report. We will call this the "report number." A file is maintained which records all reports the engine receives, with the associated report number and the time of receipt. This file is called the Global Log File (GLF) and is described in more detail later.

For each ruleset, a file recording activity within that ruleset is maintained. When a ruleset is first stated, a file for it is opened and the name of the ruleset listed at the top of the file along with the ruleset. Then, every time a report is passed to the ruleset from the buffer, the number of the report goes into this ruleset's file along with whether it was accepted by the prequalifying rules. These files are called Ruleset Log Files (RLFs), and are described in more detail later.

The log file format for both the engine log and the ruleset are similar in syntax. All numbers in the log files are strings so the they can be easily read by humans and no byte-ordering confusion is caused. All times are in Unix time. All distinct recorded items in the logs are separated by at least a new-line (i.e. "\n"). Lines starting with a '#' and blank lines are ignored. Control variable values are stored in a "variable=value" format.

2.8.2 Reconstructing Behavior From Log Files

Between the data stored in the engine's GLF and its RLFs, enough information is available to reconstruct how an engine's rulesets operated. Since a graph engine equipped with a rule-set operates deterministically, a debugger, equipped with an engine and a copy of the rules being debugged, could dump in reports one at a time to the ruleset and watch the graph grow. All reports accepted and rejected are known, as well as the order in which the graph engine saw them. Exactly how a debugging utility would present this information to a user is not relevant to the graph engine design.

Changes to control variables, as well as their initial values, for both the engine as a whole and for particular rulesets gets recorded in the respective log file. This ensures that the settings are known for all times.

In addition to reports and control variable values, the "version number" of the engine needs to be recorded in order to ensure the proper engine is run for replay purposes. However, as the code for the engine may consist of several files, a single version number will not suffice. Thus let the "version number" of the graph be an (unordered) list of files and their respective individual version numbers. The files are those that make up the engine. To run the appropriate engine, the stand-alone debugger can check out the proper versions of the various files and use those.

It is desirable to be able to be able to retain the log from multiple invocations of the engine. Furthermore it is desirable to be able to look at the log that was in effect at a given time (if any was). These points motivate the following logging architecture. A global file is maintained recording engine start-ups, shut-downs and where the logs for the invocation are stored. Lines in the file beginning with ">" are produced on the start up of the graph engine and consist of the time, and a unique identifier for this invocation (perhaps a nonce). Lines starting with "<" are produced when the graph engine shuts down (at least when it shuts down gracefully) and indicates the time.

The identifier for the invocation serves as the name of a subdirectory in which the logs for the invocation are kept. Within that directory, the log for the engine is called "engine.log" and the log for ruleset RS is called "ruleset.RS.log", for all rulesets.

2.8.3 Global Log File

The Global Log File (GLF) records reports to the graph engine across multiple invocations of the engine, records the version of the engine running and notes each time the engine is restarted, and records the control variables upon each startup of the engine as well as the midstream changes to them. There is only one GLF per aggregator dynasty (an aggregator and all successive aggregators which replace it).

As individually mentioned above, three types of records are stored in the global log file are the engine version num-

ber, control variable values, and reports. The format for the version number is exemplified by:

```
version {
    file1.pl: 1.8
    file2.pl: 1.3
    file3.pl: 1.5
}
```

Reports are recorded as the report number, the time, and the report (in the format used for communications), in that order, separated by spaces and tabs.

The grammar for the GLF follows:

```
<global-log-file> -> <GLF-event-list> <EOF>

<GLF-event-list> -> [<GLF-event> "\n"
                    <GLF-event-list>]

<GLF-event> -> <startup>|<shutdown>|
               <control-change>|<report>|
               <comment>|<ruleset-change>

<startup> -> ">" <time> ";" <invocation-number>
            "\n" <version-and-control>

<shutdown> -> "<" <time> ";"
             <invocation-number> "\n"

<version-and-control> -> "version {\n" <file-list>
                       "\n" <control-list> "\n"

<file-list> -> [<file-id> <file-list>]

<file-id> -> <filename> ":"
            <revision-number> "\n"

<control-list> -> [<control-value>
                  <control-list>]

<control-value> -> <id> "=" <id> "\n"

<control-change> -> "change: " <control-value>

<ruleset-change> -> "ruleset change: " <control-value>

<report> -> <report-number> " " <time> " " <graph>

<comment> -> "#" <text> "\n"
```

```
<EOF> = end of file marker
<filename> = any legal unix file name
<revision-number> = ascii representation of
```

a floating point number
 <time> = ascii representation of a
 unix time stamp
 <id> = the same as in communication
 protocol grammar
 <report-number> = ascii representation of
 a positive integer
 <graph> = the same as in communication
 protocol grammar
 <invocation-number> = ascii representation of
 a positive integer
 <text> = any string not containing a
 newline or EOF

< startup > is entered into the file each time the aggregator is (re)started. It stores the revision of the aggregator and the control variables under which it is running. This should only be entered upon start-ups, even though it may be tempting to dump this information periodically. (Use < control - change > if control variables change.) The first report seen by the aggregator is implicit in the file, as it is the first report logged after the < startup > is logged.

< shutdown > is entered each time the aggregator shuts down gracefully. The time of shutdown is noted. The last report seen by the aggregator is implicit in the file, as it is the last one logged before the < shutdown > is logged. (Since the engine is doing the logging, if something is logged, the engine saw it.)

< control - change > is entered each time a running aggregator has its control variables changed on the fly. It stores the new values of the changed variables.

< report > is entered each time a network report (graph) is received by the engine. It numbers and stores the report verbatim (without parsing the < graph >).

< control - value > shows a control variable and its assigned value. The < id > terminal (as defined in the communications protocol) allows for complex terms, but should not be used here for more than a variable name and a perl variable value. For the procedure to convert perl variable values to strings, see the communications protocol section.

< revision - number > is the revision number (as used by RCS) associated with the file name that precedes it.

< report - number > is a unique (across invocations of the engine) number assigned to a report. By it, reports can be uniquely identified. It is suggested (but not required) that report numbers "count" up by one as each report comes in, and that report numbers are ordered according to the order that the reports came in.

< invocation - number > is a unique number assigned to an invocation of an engine. As an engine is shut down and restarted, a new invocation number is assigned. Hence, and invocation number may be associated with the code versions and initial control variables of that invocation.

2.8.4 Ruleset Log File

When a ruleset starts up or debugging is turned on for a ruleset, a dump of the current graph space is made to the log file for the ruleset. Occasionally checkpointing the graph space, say every 1000 reports, would speed up debugging attempts at reconstructing a graph at a certain point because the reconstruction would never need to incorporate more than 1000 reports (for our example number). Also, sanity checks could be performed by incorporating more than 1000 reports and seeing if the periodic dumped graphs match those reconstructed.

The five types of records stored in a Ruleset Log File are the rules, dumps of graphs, control variable values, report acceptance, and report rejection. The rules in the log are in the format exemplified by:

```

ruleset worm_detector {
    [text for rules here]
}

```

The graph dumps are stored in the format used for communications, which is, not coincidentally, similar in structure to the above. Report acceptance is indicated by simply recording the report number and report rejection is indicated by a '!' immediately followed by the report number.

The grammar for the RLF follows:

```

<ruleset-log-file> -> <RLF-event-list> <EOF>

<RLF-event-list> -> [<RLF-event> "\n"
    <RLF-event-list>]

<RLF-event> -> <ruleset>|<acceptance>|<rejection>|
    <graphspace>|<control-list>

<acceptance> -> <report-number>

<rejection> -> "!" <report-number>

<graphspace> -> "graphspace {\n" <graph-list>
    "\n}\n"

<graph-list> -> [<graph> <graph-list>]

<ruleset> = same as in engine grammar
    definition
<report-number> = same as in GLF grammar
    <graph> = same as in communications
    protocol grammar
<control-list> = same as in GLF grammar

```

< ruleset > is entered into the file at the beginning of the file, and then each time the ruleset changes.

< acceptance > is entered every time a report passes the ruleset's prequalifying rules. The number of the report is indicated in *< report - number >*.

< rejectance > is entered every time a report does not pass the ruleset's prequalifying rules. The number of the rejected report is indicated in *< report - number >*.

< graph-space > is entered at any time, though a periodic spacing of graph-space dumps is recommended.

< control - list > is entered everytime one or more control variables are changed. The new values of the variables are indicated.

2.9 Controlling the Graph Engine

A running graph engine is controlled by the same mechanisms which are used for data sources (see chapter 4). The state variables which affect its operation include the following.

- *listen_udp*: The UDP port on which the engine is listening.
- *listen_tcp*: The TCP port on which the engine is listening.
- *department*: A department this engine is running for.
- *parent_aggregator*: The name or IP address and port for the aggregator to send reduced graph reports to. The format is *name:port* or *ip-address:port* or the empty string if there is no parent.
- *childrenname*: True if the host or department with name "name" is a child of this engine.
- *rulesets{foo}*: The graph engine rules for ruleset *foo*.
- *alert_recipients{host:port}*: True if and only if the host/port is somewhere to send alerts to. The format of the index is as in the value of *parent_aggregator*.
- *beta*: The β value to use for buffering (see section 2.5).
- *debug{foo}*: Set to *true* to start debugging for ruleset *foo*, and to *false* to stop debugging.

2.10 The Graph Engine Algorithm

To document how the engine is implemented, we provide a pseudocode version of it. The operation of the buffer and logging is ignored here for simplicity.

```

Read in rules and other parameters from configuration files
Set signal handlers to flag changes to parameters.
Initialize data structures

while(there is incoming information)
  if(rules have changed)
    dump graphs of changed rulesets
  foreach (ruleset)
    remove timed out graphs
    unless(the preconditions applied to this report are true)
      proceed to the next ruleset
    construct the new graph from the report as per the report rules
    foreach (graph in the graph space)
      unless(the new graph is adjacent to this graph)
        next
      if (the new graph corresponds to a global attribute report)
        unless (this graph contains a corresponding node instance)
          next
      foreach (node and link of the incoming graph)
        if(the combine attribute computes to true)
          add graph to temporary list
    foreach (graph in the temporary list)
      coalesce the graph data-structures
      coalesce the global attributes using global attribute rules
      coalesce auto-computed global attributes
      foreach (node and link in the overlap of the graphs)
        apply the local coalescing rules
        coalesce auto-computed attributes

```

Chapter 3

Basic Communication Protocols

3.1 GrIDS common packet format

GrIDS uses a number of different kinds of network communication. These include:

- Reports of events or partial aggregations of events (section 3.2).
- Queries for graphs (section 2.7).
- Get/Set messages to control the operation of GrIDS components (chapter 4).
- Messages to view or alter the organizational hierarchy (chapter 6).
- *Others for sure...*

This section describes aspects of GrIDS packets that are common to all types of communication.

Firstly, all GrIDS communications do occur in discrete packets. However, sometimes it is convenient to send GrIDS packets over TCP, and sometimes it may be convenient to send multiple GrIDS packets within a single UDP packet. GrIDS conventions are as follows.

All GrIDS packets, UDP or TCP, contain a header and a body. The header is any scalar, and the body is a list of scalars. Lists of any length are allowed.

The packet head serves to identify the type of packet to any GrIDS software that may receive it. The body holds the actual information. The presently allowed values for the header are shown in table 3.1. Headers other than this are not allowed in GrIDS packets.

It is convenient at times to store packets in files or print them out to display to humans. The convention adopted for doing that here is that a sequence of GrIDS packets can be stored in a file exactly as they are sent in a connection, *except* that the separator between packets will be an extra newline (character 10). This serves to make the files more readable.

A common function interface library is available to all GrIDS components to handle packaging single or multiple packets into the two formats described above. Corresponding library calls unpack such packets.

Head	Meaning
g	A Get request for a software control variable
gr	A response to a Get request
s	A Set request for a software control variable
sr	A response to a Set request
gdv	A Get request for a software control variable at a software
gdvr	A response to a gdv
sdv	A Set request for a software control variable at a software
sdvr	A response to an sdv
ghv	A Get request for a software control variable on a host via
ghvr	A response to a ghv
shv	A Set request for a software control variable on a host via
shvr	A response to a shv
r	A graph language report.
a	An upward aggregation report between engines.
q	A query to the graph engine.
qr	A response to a graph engine query.
alt	An alert generated by a ruleset.
hvr	A request for part of the organizational hierarchy.
hv	A part of the organizational hierarchy.
hve	An error in a hierarchy request.
hvu	An invalidation of part of the hierarchy.
htr	Initiating a hierarchy transaction.

Table 3.1: Allowable headers in GrIDS packets

3.2 The Graph Language

This section describes how graphs are represented in GrIDS packets. From the GrIDS engine detailed design, reports from data sources are either reports of links, or reports of attributes of links or nodes. In some cases, we need to send graphs instead of nodes and links. For example, first, when a user wants to visualize the graphs currently kept by an aggregator. Second, if we want to move an aggregator or to restart/reboot an aggregator, the aggregator should be able to request the current graphs from its child aggregators. Third, when an aggregator sends multiple reports to its parent aggregator, they can be combined and be sent as one report and hence reducing the communication bandwidth required. This will happen when, say, an event can trigger multiple rules to be fired.

We use a modified DOT graph grammar to represent graphs. Doing so allows us to use the same representation for GrIDS components to communicate and for presenting graphs to a human user. The graph grammar is shown in Section 3.2.7.

3.2.1 Labeling Nodes and Edges

We use a fully qualified domain name (e.g., olympus.cs.ucdavis.edu.) or a unique department name to label a node. The former is used to name a host, and the latter is used to name an internal node in the hierarchy. Fully qualified domain names (FQDN) are used instead of IP addresses because a host can have multiple IP addresses associated with it. What we want is a uniform way to uniquely label a host.

We use a 6-tuple to uniquely identify a TCP connection — source host, source port, destination host, destination port, sequence number of the first SYN packet, start time. If a connection is seen by multiple sniffers, they will report the same sequence number associated with the first SYN packet sent by the TCP client to the TCP server. Because TCP uses a 32-bit field for sequence numbers, it will take a long time before the same sequence number will be reused. Thus we can use the sequence number to aggregate reports from different sniffers. Moreover, we will also use the start timestamp of the TCP connection to differentiate among TCP connections that have the same sequence number. In some cases, some fields might be missing. For instance, a TCP wrapper does not know the source (i.e., remote) port number and the TCP sequence number of a TCP connection. An aggregator which receives reports with missing fields has to make some “guesses”, say based on the timestamp, to combine the corresponding reports.

We use a 6-tuple to uniquely identify a UDP “connection” — source host, source port, destination host, destination port, start timestamp, and udpid. A udpid consists of the sniffer’s id and a 16-bit integer. The latter is generated by a counter which is incremented after each use. We use timestamp and udpid to uniquely identify a UDP “connection”. By a UDP “connection”, we refer to a sequence of UDP packets that be-

long to the same transaction. It is left to the sniffer design to determine how to group UDP packets into UDP “connections”. For instance, the time the packets are observed may be used. If a sniffer observes a UDP packet and decides that it belongs to an existing UDP “connection”, the timestamp and the udpid of the first UDP packet is used to identify the “connection”.

3.2.2 Reports Containing a Connection

The following is an example report sent by a data source to an aggregator. The report shows a connection from helvellyn to jaya with the three link attributes (stime, destination port, and thumbprint).

```
digraph sniffer {
  helvellyn.cs.ucdavis.edu -> jaya.cs.ucdavis.edu
  [ctype="tcp", sport=1024, dport=25,
   stime=33311222, seq=12345,
   tpnt = "A487 239B FF72 D7A1"];
}
```

In order for an aggregator to report to its parent, the aggregator also needs to specify the rule to which the connection belongs. We use the graph id for this purpose. For example, if the connection corresponds to a worm rule (worm1), the report is as follows:

```
digraph worm1 {
  helvellyn.cs.ucdavis.edu -> jaya.cs.ucdavis.edu
  [ctype="tcp", sport=1024, dport=25,
   stime=33311222, seq=12345,
   tpnt="A487 239B FF72 D7A1"];
}
```

3.2.3 Reports Containing a Host

The following is an example report on host blanc.

```
digraph ex_mon {
  blanc.cs.ucdavis.edu [ time=23472358,
   alert="ex_monitor"];
}
```

3.2.4 Reports Containing a Graph

The following is an example report representing three concurrent connections from olympus to blanc, rainier, and sierra and a global attribute, nedges. .

```
digraph worm2 {
  nedges=3;
  olympus.cs.ucdavis.edu -> blanc.cs.ucdavis.edu
  [ctype="tcp", sport=1024, dport=25,
```

```
stime=823112009, seq=23232];
  olympus.cs.ucdavis.edu -> rainier.cs.ucdavis.edu
[ctype="tcp", sport=1024, dport=25,
stime=823112020, seq=13131];
  olympus.cs.ucdavis.edu -> sierra.cs.ucdavis.edu
[ctype="tcp", sport=1024, dport=25,
stime=823112031, seq=10101];
}
```

The next example shows how reports correspond to different rules can be combined and sent as one report. Let us assume that there is a network policy that prevents user C from connecting from olympus to sierra. In the example, "composite" is used as the graph id of the entire graph, and graphs corresponding to (possibly) different rules are represented as subgraphs.

```
digraph composite {
subgraph worm2 {
  olympus.cs.ucdavis.edu -> blanc.cs.ucdavis.edu
[ctype="tcp", sport=1024, dport=25,
stime=823112009, seq=23232];
  olympus.cs.ucdavis.edu -> rainier.cs.ucdavis.edu
[ctype="tcp", sport=1024, dport=25,
stime=823112020, seq=13131];
  olympus.cs.ucdavis.edu -> sierra.cs.ucdavis.edu
[ctype="tcp", sport=1024, dport=25,
stime=823112031, seq=10101];
}
subgraph conn_violation {
  user = "C";
  olympus.cs.ucdavis.edu -> sierra.cs.ucdavis.edu
[ctype="tcp", sport=1024, dport=25,
stime=823112031, seq=10101];
}
}
```

3.2.5 Representing Attributes on Nodes, Edges and Graphs

As demonstrated above, the attributes associated with nodes and edges and their corresponding values are stored as a comma-separated list within square brackets after the listing of that node or edge. Graph attributes with their values are listed at the top level of the "digraph" for the graph. In any case, the basic format is "name=value".

Regardless of the way a particular component represents its attributes internally, they have a common denotation in the Graph Language, referred to as flattened attribute format. This denotation depends on the type of the attribute. Recall that in GrIDS, attributes can be of type scalar, set of scalars, or list or scalars. Scalars are represented as the ascii text associated with the scalar, i.e. "abc" or "2". Sets are denoted in accordance to the common mathematical convention of being scalars, comma-separated and enclosed by curly braces,

i.e. "{a,3,b}" and "{}". The order of the scalars, of course, does not matter for sets. Lists are represented in a similar fashion, but with square brackets surrounding the scalars, i.e. "[a,c,b]" and "[3]". Whitespace surrounding scalars is significant in all three types.

However, this representation can lead to ambiguity. Consider for instance the set consisting of one member "a,b". A method is provided to encode scalars and the scalar elements of sets and lists which might be misinterpreted. "%xx" is understood to stand for character whose hex representation of the character's ASCII value is xx, i.e. "%2c" for a comma, so the above mentioned set would be represented as "{a%2cb}". The characters that must be encoded to avoid confusion are: ";", "{", "}", "[", "]", and "%"; however, any character may be encoded in this manner.

3.2.6 Quoting in the Graph Language

As in the DOT language, names of graphs, edges, nodes, and attributes as well as attribute values can be put inside of double quotes to avoid confusion with syntactic constructs of the language; however a backslash may not be used to hide the following character. Instead, a more general method is used, referred to as DOT-encoding, described a bit later. The DOT documentation does not make it clear under what circumstances a name or attribute value must be double quoted. However it is prudent to quote strings containing non-alphanumeric characters and words such as "node", "edge", and "digraph".

DOT-encoding is similar to the encoding used for scalars in the flattened-attribute format described above, however a "#" is used instead of "%" as the delimiter. That is, "#xx" represents the character whose ASCII number is xx in hexadecimal. The characters that must be encoded in this manner are "'", "\", "#", whitespace and non-printable characters. Other characters may optionally be encoded in this manner as well. Note that DOT-encoding is not an alternative for flattened attribute encoding used for attribute values. DOT-encoding is done after this encoding.

Take, as an example, the this portion of a Graph Language graph:

```
hello -> 'good#32bye' [color='ruby',
'alerts'=''{ 'tuesday's#0agone', #23#83,7}];
```

That text represents an edge going from a node "hello" to one called "good bye" with a scalar "color" attribute with value "ruby" and an "alerts" attribute which is a set consisting of the scalars: "tuesday's\ngone" (where "\n" is the new-line character), "#" concatenated with the characters whose ascii value is 131 decimal, and the scalar number 7.

3.2.7 Graph Grammar

Non-terminals are enclosed by <>. The terminals [and] are enclosed by a pair of double quotes. Graph ID's, < id > ,

is any alphanumeric string not beginning with a digit, but possibly including underscores, or a number, or any DOT-encoded double-quoted string. *< edgeop >* is --> in directed graphs. *< text >* is some flattened-attribute-encoded text.

```

<graph> -> digraph <id> {<stmt-list>}
<stmt-list> -> [<stmt> ; <stmt-list>]
<stmt> -> <node-stmt>|<edge-stmt>|
    <subgraph>|<attr-id>=<attr-value>
<node-stmt> -> <node-id>
    [ "[" <attr-list> "]" ]
<edge-stmt> -> <node-id> <edgeop> <node-id>
    [ "[" <attr-list> "]" ]
<node-id> -> <fqdn> | <id>
<fqdn> -> <id>[.<fqdn>]
<attr-list> -> <attr-id>=<attr-val>
    [, <attr-list>]
<attr-val> -> <text> | <set-val> | <list-val>
<set-val> -> "{" <text-list> "}"
<list-val> -> "[" <text-list> "]"
<text-list> -> | <text> ("," <text>)*
<node-list> -> [<node-stmt>], [<node-list>]

```


Chapter 4

The Data Source Library

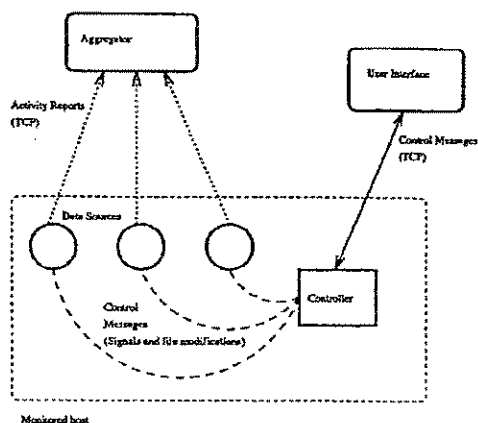


Figure 4.1: Data flow through data source modules

A GrIDS data source is a module that provides information to an aggregator. It may be such a thing as an IP sniffer or a point intrusion detection system.

Ordinary data sources must engage in two distinct forms of communication; they must provide an aggregator with graph node and link attributes, and they must respond to configuration requests. Attribute reports are sent by the data source directly to the aggregator using a protocol based on TCP. Configuration requests are sent by entities such as user interfaces using a protocol based on TCP. They are received by a Module Controller on that host, which forwards them to the appropriate data source module and then sends a response back to the querying entity. The basic configuration is shown in Figure XXXXXX.

Note that an Aggregator is, itself, a module that can be reconfigured only via a Module Controller. In contrast, a Software Manager is, itself, a module that is typically *started* by a Module Controller, but that subsequently handles its own configuration requests.

All data sources must have this functionality to work with GrIDS. A “wrapper” program may be written to manage software not written specifically for GrIDS. The wrapper must manage the software appropriately and present a compliant

interface to GrIDS.

4.1 Communication with User Interface

A user interface may communicate via a Module Controller to data source modules, to re-configure them or to query their status. Each data source carries a set of control variables. Requests from the user interface consist of reads or writes to these variables. A single Module Controller per monitored host manages all such requests.

4.1.1 Control Variables

The purpose of control variables is to exhibit and change the state of a data source module. Data source modules normally report all other information directly to their aggregators. Setting a control variable might cause a module to take action such as shutting itself down, or it might cause the Module Controller to start an instance of some module.

Control variables are named. The names may be any valid Perl variable name. Control variables may have a single value, or they may be indexed (e.g., so that an Aggregator can be dynamically reconfigured and given a *new* Ruleset.)

A set of standard control variable names will be defined. These standard variables will serve identical functions on different kinds of data source modules.

All values are represented as strings. They may contain any characters except 255. Data other than strings is represented as strings. Numerical data is represented as printable ASCII. Boolean data is represented by the strings “true” and “false”. Lists or sets are represented by indexed arrays.

The set of control variables used by any particular module is static during that module’s lifetime. It may not change without introducing a new version of that module. However, any variable name may be indexed by another string. The value of a variable at an unset index is the empty string. A variable index may be removed by setting the value of the variable at that index to the empty string.

4.1.2 Module Controller

One Module Controller will run on each GrIDS host. The Module Controller receives requests from Software Managers through TCP connections. Each request to an ordinary data source module on a host must go through the Module Controller on that host. A request to a Module Controller must specify 1) the module name of the data source module that the request addresses; 2) the module's version number [probably *not* checked by the initial implementation of the Module Controller]; 3) the *department_ID* on whose behalf that module runs; and 4) the command to be executed. Each request addresses exactly one data source module. If a set of data source modules must be addressed, a set of requests to individual modules must be issued. To minimize the number of processes, the Module Controller will also be responsible for startup of the data source modules on the host.

4.1.3 Protocol

TCP

When a Software Manager wishes to communicate control info with a data source, it opens a TCP connection to the Module Controller residing on a known port on the appropriate host. We use TCP because we do not want to drop any control packets, and we do not want to implement error detection/correction manually on top of UDP.

Although establishing a TCP connection involves some overhead, plus an open file handle, this is a small price to pay for reliable control flow.

Because the organizational hierarchy will rarely change, it may be ok for certain TCP connections to remain open indefinitely, e.g., a TCP connection from a Module Controller to the Software Manager of its own department. This allows the TCP setup overhead to be amortized over multiple commands.

The Data Source Architecture will support two basic types of commands, to Get or Set certain State Variables.

Control variable files and operation

Each data source module will have at least 3 associated files: a command file, a status/response file and an init file (it may be convenient for this initialization file to have the same format as a command file). Other associated files are optional (e.g., one or more logging files to which a module – such as an Aggregator – may write debugging info.) All associated files will comply with a *standard file naming convention*, to be defined by the implementors.

When a module controller receives an incoming command, it writes that to the command file of the appropriate data source module, then sends that process a signal (eg, USR1). Sometime after receiving that signal, a data source module will read its command file, obey the command, then update its status/response file. It then will delete its command file, to signal that it has completed its update.

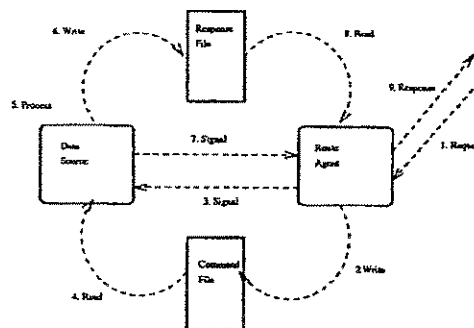


Figure 4.2: Operation of module controller

When the module controller sees the command file has been deleted, it then reads that status/response file and forwards any relevant info back along its open TCP command connection. The module controller will also set a 2-second timeout immediately after initially signalling a data source module. If the operation does not complete before the timeout, then the module controller will return an error message along the TCP connection prior to (optionally) closing that connection.

Module Controllers and Data Source Modules are intended to be single-tasking, i.e., each should process only one single command request at a time. Thus, the existence of a command file for some Data Source Module indicates that that Data Source Module is *busy* processing a request. A Module Controller should not write a new command file until the old one is deleted by the Data Source Module.

A status/response file will consist generally of State Variables and their values. It may be updated asynchronously by its data source module, or synchronously in response to a particular command. Data source modules should lock their file when writing it, so if a module controller receives a "GET-asynch" command, it will not read a module's status in the midst of an update.

The operation of the module controller is shown in figure 4.2.

Packet Format

Data source packets use the general GrIDS Common Packet Format (GCPF) described in chapter 3. The GCPF header is either *g* for a GET request, *s* for a SET request, or *gr* and *sr* for RESPONSES to GET and SET requests. The format for the body of a command or response packet will contain various fields, *separated* by a 255 character.

The *body* of a GET, SET, or RESPONSE packet will contain the following fields:

- SET | GET | GET-asynch | OK | ERROR-text | WARNING-text
- Data Source Name

- Version # (minimum acceptable version; assume upward-compatible) (RESPONSE packet may insert *actual* Version #, or more likely may ignore this field entirely)
- Department ID represented by that module.
- StateVarName
- Value
- StateVarName
- Value
- StateVarName
- Value

For GET requests, the *Value* fields may be empty, in which case there may be adjacent 255 characters.

The file format of the command file and the status/response file has this same form, although each character-255 separator is followed by a newline for readability.

If a packet is a RESPONSE to a GET command, then *all* the requested StateVarNames and corresponding Values will be returned, unless there was an error outcome. If this is a RESPONSE to a SET command, then *none* of the original StateVarNames and corresponding Values will be returned, unless an error occurred. In that case, the response packet may include only those StateVar/Value pairs that help explain the error.

Note that an *ERROR-text* value (pertaining to the entire RESPONSE) may appear in the initial field, in addition to other *ERROR-text* fields which may appear as Values of StateVars. Possible *text* for ERROR fields include:

- Failure,
- Timeout,
- No such data source,
- No data source here representing specified Department,
- Bad version number,
- Set failed (invalid value),
- Set failed (read-only variable),
- Set/Get failed (no such StateVarName).
- Set/Get failed (unauthorized access).

4.1.4 Special Aliases & Subcommands to Module Controller

Among the set of *required* StateVars, we have reserved one special StateVar name as an alias, "*ALLSTATEVARS*".

(The initial implementation may ignore this feature.)

A command to GET *ALLSTATEVARS* for some module should return a RESPONSE containing the StateVarNames and Values of all its scalar StateVars; for its indexed StateVars, only the *indices* themselves will be returned within a single Value field, enclosed in squiggly brackets.

A command to GET *Ruleset {ALLSTATEVARS}* for some module should return all Rulesets.

A command to GET *ALLSTATEVARS {ALLSTATEVARS}* for some module should generate an exhaustive enumeration of both scalar and indexed StateVars.

By addressing a SET or GET command to the module controller itself, certain subcommands can be represented.

A command to the Module Controller to GET *INVENTORY* would cause the Module Controller to read its *Task File* and return a list of every GrIDS module currently running on its host.

We anticipate implementing SET subcommands to START or KILL (a particular module), and to add or delete entries in our STARTUP_TASK_FILE (via subcommands TASK_FILE.ADD and TASK_FILE.DELETE).

We illustrate the packet format of a SET subcommand by showing how a new instance of a module is invoked. To STARTUP a new module, send a SET command to the appropriate module controller. It should specify the following named fields (as pseudo control variables) and values:

- *module* == "module.controller" (Address it to the module controller.)
- *mc.command* == "START" (Tell module controller its sub-command.)
- *mod.module* == Official/generic name of module you want to start. This might *not* be same as its executable pathname.
- *mod.department* == official dept_name the new module represents.
- *mod.version* (If you know it. Ignored in first implementation.)
- *report.to.host.port* == (optional) A string of form 'host:port' so the new module can locate its parent or significant other. (Defaults to the host:port of the aggregator for that module controller's department.)
- *machine.ad* == (optional) host:port of whoever is authorized to issue subsequent SET/GET cmds to this module. (Default is host:port of module controller's parent.)

- *init_file_name* == (optional) init file for new module to read.
- *path_prefix* == (optional) file space work area for the new module.

If either of the last two fields are null, the module controller will load them with whatever defaults (perhaps null) come from its `CONSTRAINT_TASK_FILE`.

4.2 Data Source To Aggregator Communication

Reports from the data source modules are sent to the Aggregators asynchronously via TCP. Each data source module reports to one Aggregator, while an Aggregator gathers information from one or more data source modules.

The host name and port number of the appropriate Aggregator are specified in a data source module's state variables. The state variables are set initially from the config file at the startup time of the module.

Each data source module has a set of state variables that control the amount and type of reports to be sent to the Aggregator. For example, a Sniffer module might have a state variable called `FROM_HOST`. When the variable is assigned with the value of a certain host, say Rainier, the sniffer would report only those connections initiated by Rainier.

The reports from data source modules are expressed in DOT format. For example, a host is represented as a node, a connection is represented as an edge, and attributes could be attached to the nodes and edges. Refer to the "Grids Communication Protocol Detailed Design" for details about representing information in the DOT format. Reports are passed as strings through the TCP connections.

4.3 Library Functions

A set of library functions will be developed for the User Interfaces, Module Controllers, Software Managers, and Data Source Modules. Library functions for the User Interfaces include,

- `set_command(Module_Controller, command);`
- `get_command(Module_Controller, command)` which returns the information requested.

The library functions for the Module Controller include,

- `write_command_file (data_source_module, department.ID, command);`
- `write_config_file (data_source_module, command);`
- `read_config_file (data_source_module);`
- `read_status_response_file (data_source_module, department.ID, StateVarNames);`

The library functions for the data source module include,

- `read_config_file();`
- `sig_handler()` which handles signals from the Module Controller.
- `read_command_file()` which returns the command read;
- `write_status_response_file` (in addition to writing out the Values of all StateVars, this will update the required `LAST_UPDATE_TIME` before writing it out);
- `delete_command_file()` which indicates it has finished that command;

A user communicates with the data sources modules using the library functions as provided for the User Interface.

Each data source module should support a standard set of "read-only" state variables that describe the execution of the module. They include,

- `running`; which is either True or False;
- `PID`;
- `LAST_UPDATE_TIME`;
- `module_name`;
- `version_number`.
- `department.ID`;

Each data source module also must support a standard set of required state variables to control the execution of the data source module. They include,

- `execute_config_file`;
- `shutdown`;

4.4 Detailed Design of Module Controller

4.4.1 Startup and Trust Issues for Module Controllers

At startup, the Module Controller reads its own config files to determine:

- parent Software Manager (Host:Port)
- Aggregator to whom Data Source Modules report (Host:Port of Aggregator itself)
- Organizational Hierarchy Server (Host:Port)
- timeout period (to wait for local modules to respond to reconfigs)

Liveness Protocol between Module Controller and Software Manager

The uptime/liveness protocol between a Module Controller and its parent Software Manager may be initiated by either party. Each party should identify itself, and assert its kinship relationship with the other. This protocol may cause a Module Controller to question its parentage (rightly or wrongly). If a Module Controller believes an unauthorized Software Manager parent is trying to claim it, or if its overtures are rejected by its purported parent, then the Module Controller should contact the Organizational Hierarchy Server. This method will handle cases where a host (and its Module Controller) is down, and misses the notification that it is being moved to a different department (Software Manager), and henceforth should obey commands from a different parent Software Manager.

Normally, the parent Software Manager will initiate the uptime/liveness protocol by sending a request to *GET Module.Controller.Parent*. The Module Controller – if alive – should respond with the Value of its corresponding StateVar, unless the request is from an unauthorized Host:Port, in which it returns the appropriate ERROR message.

Upon startup, a Module Controller will wait to be contacted by its parent Software Manager. If its parent has not claimed it within some timeout, then the Module Controller will initiate the uptime/liveness protocol, by sending a RESPONSE to a nonexistent request to *GET Module.Controller.Parent*. If the parent is alive and willing to assert its kinship relationship, then it initiates the normal uptime/liveness protocol. This second form of the protocol makes sense in cases where a Module Controller wakes up long after it has been abandoned by its former parent during an organizational change (organizational *downsizing* leads to family breakup :-)

4.4.2 Special Files for Module Controller

The Module Controller has 2 special config files: Its *Task File* contains commands to invoke (potentially multiple) instances of certain programs, with appropriate parameters (some of those parms may be variables, e.g., all dumb Data Sources must be given the Host:Port of this department's Aggregator). This config file is writable by the Module Controller, since it represents a dynamically changing sequence of tasks the Module Controller should execute each time it awakens.

The Module Controller can read – but not write – its *Constraint File*, its second special config file. This file will map the *generic name* of a GrIDS module (as it would appear in a *Task File*) to a specific executable path.

The *Constraint File* may contain *patterns* that further constrain what can appear in the *Task File*, e.g., command-line parameters. In this manner, we can ensure that pathnames and parameters in the *Constraint File* comprise “safe” commands, yet also allow the Module Controller to dynamically invoke new instances of programs.

(The Module Controller also needs a file to store the PID and config-file path prefix corresponding to each GrIDS module running on its host. It is not sufficient to store these in RAM, because whenever the Module Controller awakens, it must ascertain whether some of its previous children are still alive, to avoid invoking redundant instances of those tasks. This file should also store the department ID that “owns” each task. This file may be a third file, or it may combined with the *Task File*.)

The main loop of the Module Controller receives a TCP connection and/or packet, decodes the incoming command, decides whether to allow access (based on the source Host:Port), processes the command, and responds to the sender.

4.4.3 Machine Access Control

Most commands are intended for “ordinary” Data Source modules. By checking the Host:Port of an incoming TCP connection, a Module Controller will allow only its own parent Software Manager to access (via GET or SET) StateVars on an ordinary module. (A Module Controller knows its parent Software Manager via its own StateVars.) Two types of special modules require different access control.

If a module is a Software Manager or an Aggregator, then it may be running on behalf of another department – ie, not the dept on whose host it is headquartered. In that case, an Aggregator should trust commands from the Host:Port of its own dept's Software Manager, and a Software Manager should trust commands from the Host:Port of its own Software Manager parent. In general, these may be different from the local Module Controller's Software Manager parent.

Thus, each instance of an Aggregator or Software Manager must be associated with an ACL specifying which Host:Port is allowed to reconfigure or shut it down. (The initial implementation will allow an ACL to contain only a single Host:Port. This should be sufficient.) The ACL will be implemented as a StateVar for its corresponding module.

The Module Controller will check that ACL before allowing Write access to an Aggregator. The Module Controller's own Software Manager parent is *implicitly* granted Read access to every module running on that host. Commands to GET or SET from other Host:Port origins are unauthorized, and should generate an error message as a response.

4.4.4 Access Control for Special Modules

It is the responsibility of a Module Controller to perform this enhanced access-control checking when it receives commands directed at an Aggregator module. Thus, to perform *authorized* reconfiguration of a resident-alien Aggregator, the Module Controller must be able to read (and write) a file containing the Aggregator's ACL.

In contrast, since a Software Manager will reconfigure itself, it performs the access-control check itself (the Module Controller is *not* involved in the protocol at all). Although

the local Module Controller does not need to read a resident-alien Software Manager's ACL file, it *does* need read access to certain info in the Software Manager's config file.

This is because a Module Controller's own Software Manager parent has the "right" to inventory all modules running within its domain. Hence the Module Controller needs *read* access to config files of resident-alien Aggregators and Software Managers, so it can determine who they represent. (This capability allows a Module Controller to return a helpful error message if a User Interface tries to delete its host, yet a foreign Software Manager is headquartered on it.) Note that a Module Controller's own Software Manager parent is not allowed *write* or *shutdown* access to a resident-alien Aggregator or Software Manager that is already running.

Finally, who should be authorized to start a new Software Manager or Aggregator dynamically? A Module Controller should obey incoming commands from its *own* departmental Software Manager to start a *new* Aggregator or Software Manager.

But after launch, write-access to a new Aggregator or Software Manager should be allowed only from whatever Host:Port was specified in its startup command (and stored in its ACL file). In this manner, a native Software Manager authorizes the residency of a foreign module. (By setting strict constraints on the invocation parameters of a Software Manager in the special config file, a department can prevent certain foreign departments from headquartering their Software Managers or Aggregators on its host.)

4.4.5 Module Controllers Should Use Standard Port

Humans will attempt to ensure that all Module Controllers occupy the *same* known port on all hosts. This value will be loaded by hand into each host node in the Organizational Hierarchy file. When a Module Controller must occupy a non-standard port, it is the responsibility of human operators manually to update the Organizational Hierarchy file to reflect that situation.

4.4.6 Deleting Hosts Running Critical Modules

When a host is to be deleted, we already specified this operation should fail (with an informative error message) if the host is running any Aggregator or Software Manager. Thus, the Module Controller must know every GrIDS module it is running, including resident-alien. Because the Module Controller has *implicit* Read-access to resident-alien's config files, this allows the Module Controller to inform a User Interface *whose* Aggregator is causing an error in response to a host-delete command.

4.4.7 What Software is Executing?

The Module Controller will have a known set of StateVars. A request, e.g., to GET *INVENTORY*, would cause the Module Controller to read its *Task File* and return a list of every GrIDS module currently running on its host.

4.4.8 Launching Recurring Modules

The Module Controller will have a known StateVar, e.g., *CHRON_JOBS*, which will be indexed by the frequency (in minutes) at which to launch a task, and the particular task to launch.

This will *not* be implemented initially!

4.5 Initialization of Modules

When launching a new task, the Module Controller will pass it a command-line argument indicating the pathname of its initialization/config file (which may or may not be unique to this task), the *path prefix* of its unique control files, plus optional additional args (e.g., department ID).

The new Module will construct the exact pathnames of its unique control files (command, status_response, and log_debug files) by appending a known set of suffixes to the given prefix.